GRAPHIC DISPLAY AND MANIPULATION OF TREE-TYPE DATA STRUCTURES

Claude Myrick Holifield

United States
Naval Postgraduate School

THESIS

GRAPHIC DISPLAY AND MANIPULATION
OF
TREE-TYPE DATA STRUCTURES

by

Claude Myrick Holifield, Jr.

December 1969

Graphic Display and Manipulation of Tree-Type Data Structures

by

Claude Myrick Holifield, Jr.
Major, United States Marine Corps
B.S., Auburn University, 1961

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1969

ABSTRACT

The subroutine package TREEPAK is designed to interface a PL/I program with a graphic display unit, allowing display, creation and manipulation of tree structures, while providing the user with a graphic representation of the tree. The package is implemented under IBM System/360 using the IBM 2250-1 graphic display unit as the primary input/output device. Facilities include creation of trees from the alphanumeric keyboard and cards, adding and deleting subtrees in existing trees, and saving and retrieving trees. Trees may be passed between TREEPAK and the user written main program. An implementation of the Wang Algorithm is given as an example application program.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. STATEMENT OF THE PROBLEM

The objective of this thesis was to develop an interactive subroutine package which interfaces a PL/I program with a graphic display unit, allowing display, creation and alteration of tree structures in graphic form. There is a large class of problems in which a hierarchical relationship exists among the data, and a tree type data structure is often an efficient and effective means of representing this relationship. This package, called TREEPAK, is designed to provide the capability of on-line input and manipulation of data in tree structures with the additional facility of allowing the user to view a graphic representation of the tree. TREEPAK is quite general and may be accessed from any PL/I program. Figure 1 illustrates the general configuration of a system using TREEPAK.

Various terminology has been used to characterize trees and the nodes of a tree. The terminology used throughout this thesis is described in order to clarify the discussion of TREEPAK.

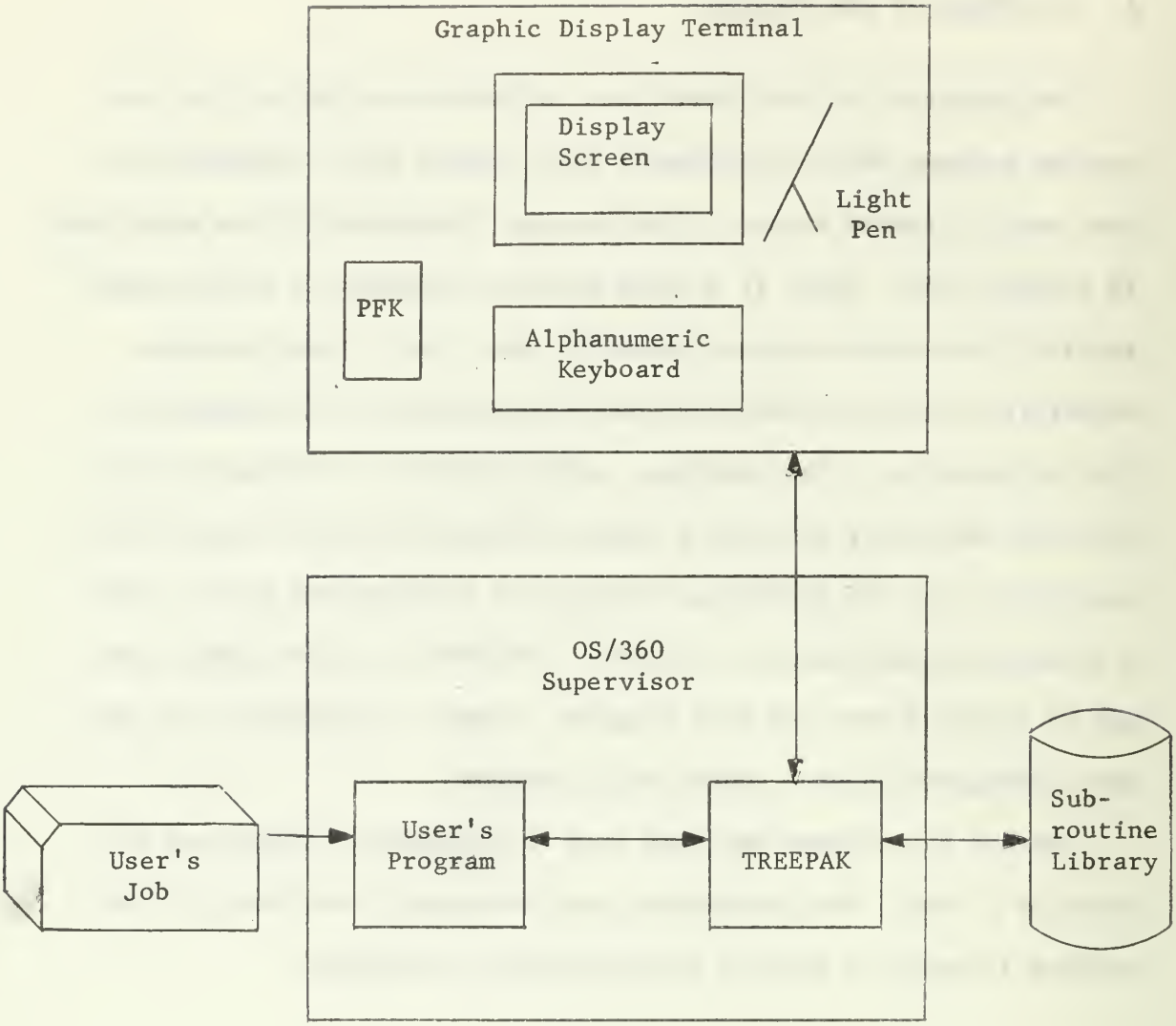


Figure 1. General Configuration

B. TERMINOLOGY

A tree has been formally defined [1] as a finite set T of one or more elements, called nodes, such that (a) there is a distinguished node R called the root, and (b) the remaining nodes (excluding the root) are partitioned into $m \geq 0$ subsets $T_1, T_2 \ldots T_m$ where each of these subsets is in turn a tree. Although no relationship between nodes is expressed by this definition, it is assumed that the root of any particular tree is logically related to the remaining nodes.

The trees $T_1, T_2 \ldots T_m$ are called the subtrees of R. This definition is represented graphically in Figure 2, which shows a tree with ten nodes. The root A has three subtrees whose roots are B, C and D. B, C and D have one, zero and two subtrees respectively.

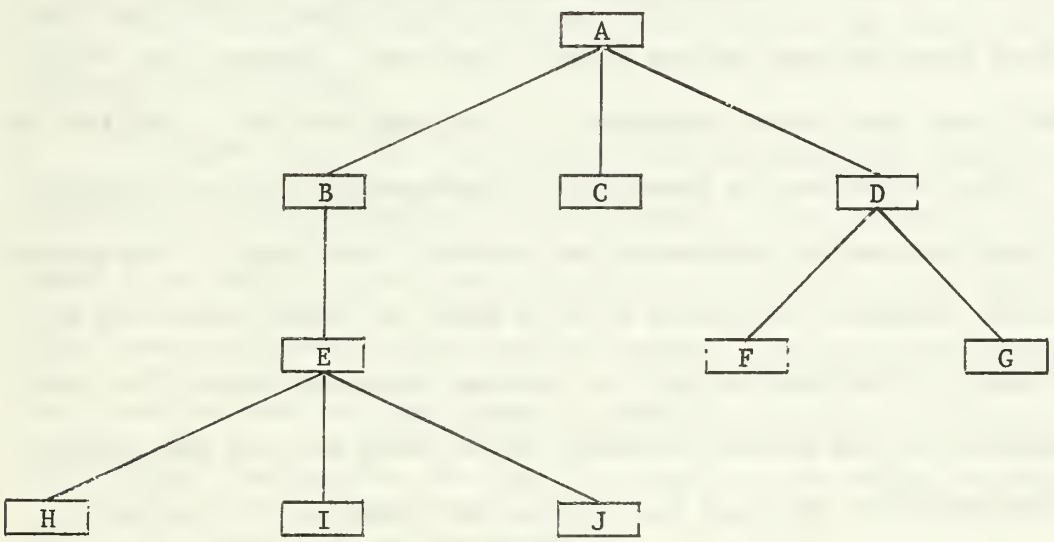


Figure 2. Graphic Representation of a Tree.

The roots of the subtrees of a node are called the successors of that node. A node is the father or predecessor of its successors. The number of subtrees of a node is its degree. Thus the degree of A is three, the degree of B is one, and C has degree zero. A node which has no successors is called a terminal node.

The level of a node with respect to the root is defined by the following: the root is at level one, and the level of every other node is one plus the level of its father. Thus B, C and D are at level two; E, F and G are at level three. Although these definitions apply to graphic trees as shown in Figure 2, they may be applied to tree structures used in list-processing as well.

A familiarity with list-processing fundamentals is assumed, however, a presentation of these concepts is found in references 1 and 2. It should be noted that the essential difference between a tree and the more general list is that a tree does not contain cycles, i.e., there is a unique path from the root to any node in the tree. However, the "sublists" of lists need not be disjoint. A list may even be a "sublist" of itself. This difference is important in application, but in a graphic display any list may be represented as a tree if some node is designated as the root. Further discussion of this point is found in Section II.C.

As used in list-processing, the internal representation of a tree node consists of some number of memory cells which contain the information associated with the node and a means for locating its successors or its predecessors, or both. The content, format, and representation of the information is entirely dependent on the application. The nodes may contain no information at all, e.g., the non-terminal nodes of a LISP tree. In a hierarchical data retrieval system each node may contain

large amounts of coded data in various formats. The information associated with the node will hereafter be referred to as the name of the node, since the information may be thought of as an interpretation of the name.

A means for locating the successors or predecessor of a node is required for traversing the tree, i.e., visiting the nodes in some desired order. The following recursive algorithm is commonly used for traversing trees: (a) visit the root, then (b) visit each of the successors of the root. In an application which uses this algorithm for traversing trees there must be a means for locating the successors of a node. Frequently this is provided by pointers contained within the node. In this context a pointer is simply a number which represents the address in memory of a successor, or "points to" the successor. Other representations of trees are found in references 1 and 2. The most suitable representation depends primarily on the order in which the nodes are visited.

The algorithm given in the preceding paragraph for traversing a tree reflects the recursive nature of trees. This recursive nature comes from similarity of substructures. As stated in the definition of a tree, each node is the root of a tree. This property allows each node to have the same format in memory regardless of its degree.

In the next section two specific applications which use tree-type data structures are described.

C. BACKGROUND

Tree-type information structures arise in many computer applications. Of particular interest in the development of this thesis

were those applications which use tree structures in interactive environments. The Transformational Grammer Tester (TGT) at Systems Development Corporation [3] is one such application. Transformational grammers are built on the concept of the logical separation of two types of grammatical structure, the deep and surface structures. Accordingly, there are two systems of rules associated with the syntactic component of the grammer. Phrase structure rules generate deep structure taking the form of labeled trees, and transformational rules map trees to other trees determining the ultimate surface structure of a sentence. As the grammer becomes large and the linguist attempts to account for more phenomena in the language he is describing, it becomes more difficult to provide for all the interrelationships of the rules. TGT was designed to assist the linguist in analyzing, refining, and extending the grammer initially specified.

The most important tasks performed by TGT center around its ability to execute transformations. Using TGT, the linguist can determine the applicability of transformations, execute them, and display the results. Facilities are available for creating, displaying, and manipulating trees which represent the structure of a sentence much like the facilities available in TREEPAK. However, the graphic display unit is only an auxiliary output device in the TGT system. Input is through a teletypewriter terminal, thus, all commands and operands must be typed, as opposed to quicker and more natural means available when the graphic display unit is the primary input/output device.

J. H. Bennett and others describe a series of computer programs developed at Air Force Cambridge Laboratories [4]. These programs, called SAM, are experimental tools for studying techniques in theorem

proving using man-machine interaction. Given a finite set of logical formulas, SAM attempts to generate "interesting" consequences by using four processes called reduction, expansion, digression, and contradiction. Reduction uses a set of formulas from the original set to reduce or simplify a given formula using logical rules of the Predicate Calculus. Expansion and digression use these same rules to generate new formulas from the initial set of formulas. Contradiction attempts to eliminate "trivial" formulas by finding a contradiction of the negation of a formula. SAM applies these four processes in a pattern which allows the newly generated formulas to stay in the set only if they cannot be reduced by reduction or eliminated by contradiction.

Using the graphic display unit as the primary input/output device, the user initiates action by setting up an initial list of formulas. SAM then starts to generate consequences of these formulas. As SAM works on the lists of expansions and reductions, the user is able to watch these lists on the display. Each new formula generated appears on the display and each formula eliminated disappears as SAM updates the current list of formulas. The user may intervene at any point in the process. If SAM is running out of useful things to do with the formulas first given, the user may insert additional formulas. He may guide the process by deleting formulas which seem to be unimportant or distracting.

Some of the initial formulas may be marked by asterisks to indicate that they are the negations of formulas whose proofs are being sought. In this case, all the consequences of the original starred formulas are starred. It is hoped that SAM can derive a contradiction. If so, SAM has demonstrated by contradiction that the disjunction of the unnegated versions of the original starred formulas is a

logical consequence of the other initial formulas (axioms and theorems). The authors report that one result of their experimentation has been the actual solution by man-machine interaction of an open problem in Lattice Theory.

The significance of systems of this type seems to be that by using man-machine interaction, results may be obtained which neither could obtain alone. The two areas mentioned, language analysis and theorem proving, are still in the early stages of their development and complete algorithmic processes are not yet known. It appears to be particularly advantageous for applications of these types to have the capability of human intervention and guidance when necessary, thus taking advantage of the speed and logical power of the computer, as well as the insight and experience of the human. When the application also uses a hierarchical data structure, it seems that the ability to view and interact with the structural relationships in graphic form would enrich the interactive process.

In the next section a description of TREEPAK is given. The following section describes an example application program which allows the user to interact with an implementation of the Wang Algorithm using TREEPAK.

II. DESCRIPTION OF TREEPAK

A. GENERAL OVERVIEW

TREEPAK, which consists of approximately nine hundred PL/I statements, is designed to provide any PL/I program the capability of displaying and manipulating tree structures. PL/I was chosen as the implementation language for two reasons: (a) general list processing features are available in PL/I, and (b) recursive subroutine calls are allowed. As previously stated, algorithms for traversing trees are inherently recursive in nature and the availability of a language capable of recursion is a significant asset in programming.

The IBM 2250-1 Graphic Display Unit is the terminal input/output device for TREEPAK. This unit is equipped with an alphanumeric keyboard, light pen, and programmed function keyboard (PFK). All three of these facilities are utilized to perform various functions. Maximum use is made of the light pen and PFK since their use is more natural than using the alphanumeric keyboard. The alphanumeric keyboard is used only when it is necessary to type a name or a list.

The interface between TREEPAK and System 360 graphics routines is provided by IBM Contributed Program 360-00.6.009 (WJS999) /5/. This subroutine package provides access to the graphic input/output routines and Graphics Problem Oriented Routines included in references 6 and 7. The package also allows interrupts from the console to be handled with PL/I ON CONDITIONS.

B. ENTRY AND EXIT CONDITIONS

TREEPAK is designed to run as a subroutine to a user-written main program. Two arguments are required in the call to TREEPAK: a PL/I pointer variable and a character string. If the value of the pointer variable is not the NULL pointer it is assumed to point to the root of a tree which exists in the main program at the time of the call. The tree and the character string are both displayed on the screen as a result of the call. After TREEPAK is entered all of its capabilities are available to the user, including creating and displaying new trees, saving the tree currently on display, and altering the tree on display. A detailed account of the capabilities is given in the next section. When the user returns control to the main program a pointer to the root of the original tree is returned.

The required format of the nodes of the tree is shown in Figure 3. This format is a direct result of the following PL/I structure declaration:

```
DCL BOUND FIXED BINARY,
    1 NODE BASED (P),
      2 NAME CHAR(10),
      2 DEGREE FIXED BINARY,
      2 BRANCH(0:BOUND REFER (DEGREE));
```

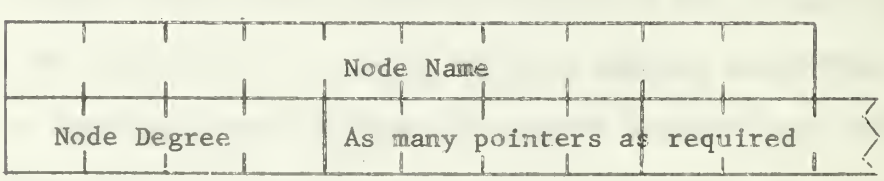


Figure 3. Node Format

The first ten bytes contain the name of the node. These ten bytes must be EBCDIC coded alphanumeric characters. The next four bytes contain a fixed binary integer which is the degree of the node. The node degree is followed by four-byte pointers to the successors of the node. These pointers contain the absolute internal address of each successor. The nodes are declared as PL/I based structures /‾8 and 9_‾/.

The NULL pointer may be used as an argument in the call to TREEPAK. In this case no tree will be displayed until one is created by the user. Hence, there is no requirement that a tree exist in the main program at the time of the call. If TREEPAK is entered with the NULL pointer, a pointer to the tree on display is returned to the main program. This allows the user to create a new tree while in the subroutine, manipulate the tree, and pass it back to the main program for analysis.

C. CAPABILITIES

The capabilities for operations on trees include creating new trees, adding a structure to an existing tree, and deleting all or part of a tree. Operations are performed on the "current" tree, i.e., the tree currently on display. The following sections contain a general description of each of these operations. Detailed instructions on how to perform each operation are given in Appendix A. Before continuing a discussion of the capabilities, a description of the display screen is given.

The tree being displayed is centered on the screen, with equal vertical distance between each level. All nodes on the same level appear on a straight line equidistant across the screen. This spacing takes maximum advantage of the display area of the screen since

the tree appears as large as possible for that size tree. Each node is numbered, with node names corresponding to each number listed along the sides of the tree. The list notation described in the next section is shown across the top of the screen. Messages to the user, if applicable, appear on the bottom line of the screen. See Appendix C for photos of typical displays.

If the current tree has many levels, or many nodes at some level, it may not be possible to display all of the tree at the same time. To do so would make the screen completely unreadable. Thus, a maximum of ten levels and twenty nodes at each level is displayed. In this case some nodes have successors which do not appear on the screen. These nodes are marked with a "+" on the display. The substructures of these nodes may be displayed separately (see Appendix A, PFK key 4).

1. Creation of Trees

Trees may be created either by reading them from cards in the input stream or they may be typed at the console using the alphanumeric keyboard. The tree in Figure 2 could be created by typing the following string of symbols:

(A(B(E(H,I,J)),C,D(F,G))),

The rules for this notation are: (a) successors of a node are parenthesized immediately following that node, (b) nodes on the same level are separated by commas, and (c) the entire list is enclosed in parentheses. This notation is completely general and may be used to unambiguously describe any tree in which all nodes are named. Only one node in a tree appears at level one. TREEPAK only creates one tree at a time,

thus care must be taken to ensure that the expression contains exactly one node at level one. If the expression represents more than one tree, only the first tree in the list will be created.

2. Altering the Tree Structure

The PFK and light pen are used for manipulating the tree on display. Operations are performed on the "identified" node. To identify a node for subsequent alteration, the light pen is activated on the node. Upon sensing that a node has been touched with the light pen, TREEPAK places a visible pointer beside the node on the screen to signal to the user that the proper node has been identified (see Figure 4, node number 3). If a node has not been identified in this manner, an attempt to alter the structure will result in no action. After a node is identified, two basic operations may be used to alter the structure -- add and delete.

The add function performs the operation of adding a node or a structure as a new successor to the identified node. When the user issues this command he is then given the option of adding the new structure in any position he wishes. Figure 4 shows node number 3 as the identified node after issuing the add command. The position of the new successor is chosen by placing the light pen on the asterisk which is in the desired location. The new successor may be created from the alphanumeric keyboard, or it may be a tree previously saved in the save area described in paragraph 3.

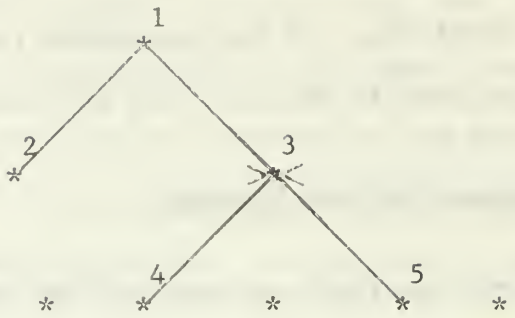


Figure 4. Adding a Successor to the Identified Node

The identified node and its substructure may be deleted from the tree. The detached portion of the tree may be named and saved for further use, if desired, or it may be deleted from the system. The next paragraph contains a discussion of saving trees in the save area and retrieving them.

3. Naming and Saving Trees

The tree on display, or any part of it, may be saved for later reference. If a subtree of the current tree is saved, it is not necessary to detach that subtree, although it is possible to do so as discussed in the previous paragraph.

When placing a tree in the save area, the user may assign it a name. If a name is given which already exists in the save area, TREEPAK informs the user and replaces the old tree. Each tree saved is also assigned a number. The user is informed of this number and later references may be by name, if one was assigned, or by the assigned number. Saved trees may be recalled at any time, either to become the new current tree, or for addition to a specified node.

A list of the names and numbers of trees in the save area can be displayed. If desired, trees may be dropped from the save area. See paragraph II.D.5 for further information about the save area.

D. IMPLEMENTATION

Some of the techniques used in implementing TREEPAK are discussed in this section. The details of the coding are omitted here (the program listing is found in Appendix B).

1. Coordinates for the Display

To display a tree, the position of each node on the screen must be computed. The screen can be thought of as a Cartesian coordinate grid with the lower left corner of the screen assigned coordinate (0,0) and the upper right assigned coordinate (4095,4095). To compute the coordinates for each node in the display, the tree to be displayed must be traversed twice. On the first traversal, the number of levels and the number of nodes at each level to be displayed are counted. These numbers determine the distance between nodes on the screen. On the second traversal, the coordinates for each node are computed, using the information obtained on the first traversal and the node's position in the tree. The following formulas are used for computing the coordinates of each node on the screen. H and V are the horizontal and vertical distance, respectively, between nodes. X and Y are the horizontal and vertical coordinates.

```
H = 4096/(No. of nodes at that level + 1)
V = 4096/(No. of levels + 1)
X = H(No. of nodes already visited at that level + 1)
Y = 4095 - V(Level of the node)
```

These computations are performed by TREEPAK each time a tree is displayed or altered. The number of levels and the number of nodes at each level are saved for use in locating the identified node as described below. From these numbers the actual node position and the vertical and horizontal distances between nodes can be computed at any time.

2. Cycles in the Tree

From the definition of a tree given in Section I.B, a tree does not contain cycles, i.e., there is a unique path from the root of the tree to each node. In practice it is often useful for list-processing trees to contain cycles. The structure may still be pictured as a tree by repeating all overlapped nodes until none are left. Alternatively, the first node in a cycle may be treated as a terminal node on the second and subsequent visits. The former method produces an infinite tree if the tree is recursive. Thus, the latter method is the only practical way of implementing cycles in a graphic display.

To determine when a cycle is encountered, each node must be marked as the tree is being traversed. Each time a node is visited, a test is made to determine whether the node has been marked. If so, then a cycle has been found. TREEPAK marks the nodes by changing the sign of the node degree. When a node is found which has a negative degree, this node is treated as a terminal node for the display, except that a "%" is placed beside the node on the screen and the user is informed that the tree contains cycles. Changing the sign of the degree introduces the additional requirement of resetting the degrees to their original value after each traversal.

3. Finding the Identified Node

When the light pen is activated on a node, the coordinates of the location at which the light pen is detected are used to determine which node is the identified node. The level of the node can be determined using the distance from the top of the screen to the point where the light pen is detected. Referring also to V from paragraph 1,

Level = (distance from top of screen to detect)/V.

The number of nodes on the left of the identified node at the same level is found using the distance from the left of the screen to the light pen detect, along with H from paragraph 1 as follows:

No. on left = ((distance from left of screen)/H) - 1.

These numbers uniquely describe a node in the tree. The tree is then searched until the node in that position is found, and a pointer to that node is saved.

4. Altering the Structure

Because of the internal representation used for the nodes, the amount of storage required to contain a node depends on the degree of the node. Altering the tree structure involves adding or deleting a successor, which changes the degree of the affected node. Thus, when the structure is changed, the node whose degree is changed must be replaced with a new node of the appropriate size. Figures 5 and 6 show how TREEPAK handles additions and deletions. The number alongside each node represents an arbitrary memory address for the node. The organization of the nodes is as shown in Figure 3.

Figure 5(a) shows a tree of five nodes. Figure 5(b) shows the tree with node E deleted. Note that in Figure 5(a) the cell shown at memory address 140 is in the free storage area. This cell is obtained from free storage using the PL/I ALLOCATE statement. The proper information from the old node B is placed into it. The pointer to B in node A is changed to point to the new cell. The old node B at address 100 is released to free storage using the PL/I FREE statement.

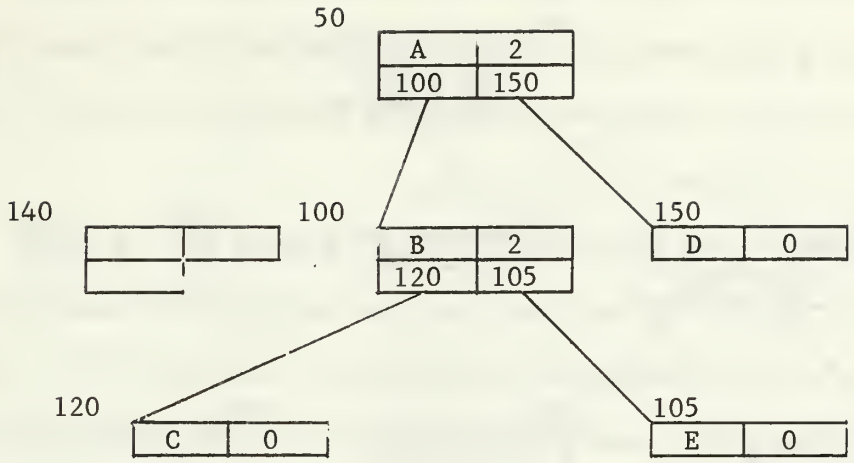
Figure 6(a) shows the same tree as Figure 5(a). Figure 6(b) shows the tree after the addition of node F as a new successor to node D. In this example, the cell at address 190 is obtained to hold the new node. The cell at address 150 is released to free storage after the replacement has been made and the appropriate pointer in node A changed.

5. The Save Area

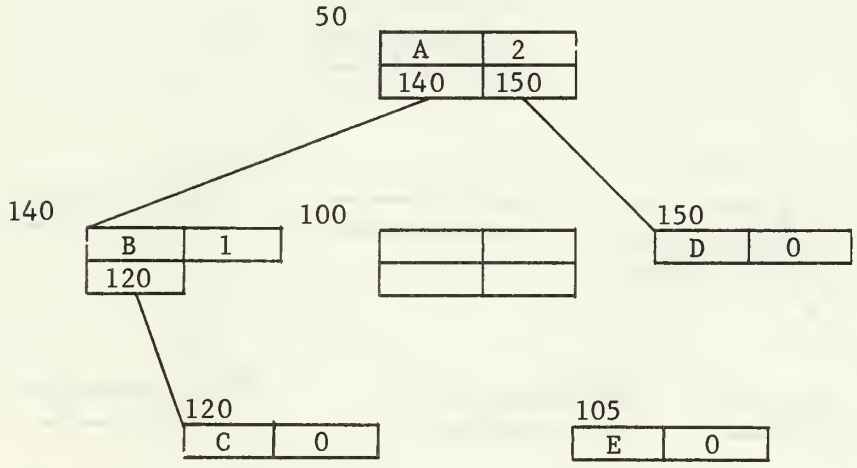
When trees are saved, only two items of data are recorded in the save area -- a pointer to the root of the tree and the name of the tree. The save area consists of the following PL/I structure declaration:

```
DCL TOP FIXED BIN,
    1 SAVE BASED(P),
      2 NUM FIXED BIN,
      2 ROOTS(0 : TOP REFER (NUM)),
        3 PNTR POINTER,
        3 NAME CHAR(10);
```

NUM is the number of trees which are currently saved, determining the upper bound for the dimension of ROOTS. If no name is assigned to a tree its NAME is blank. The number assigned to the tree by TREEPAK corresponds to the subscript of ROOTS which contains the pointer to the tree.

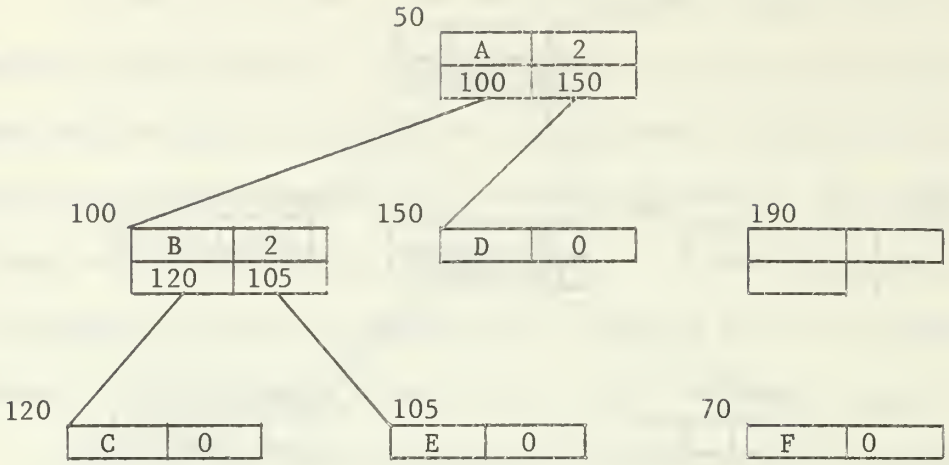


(a) Before deleting node E

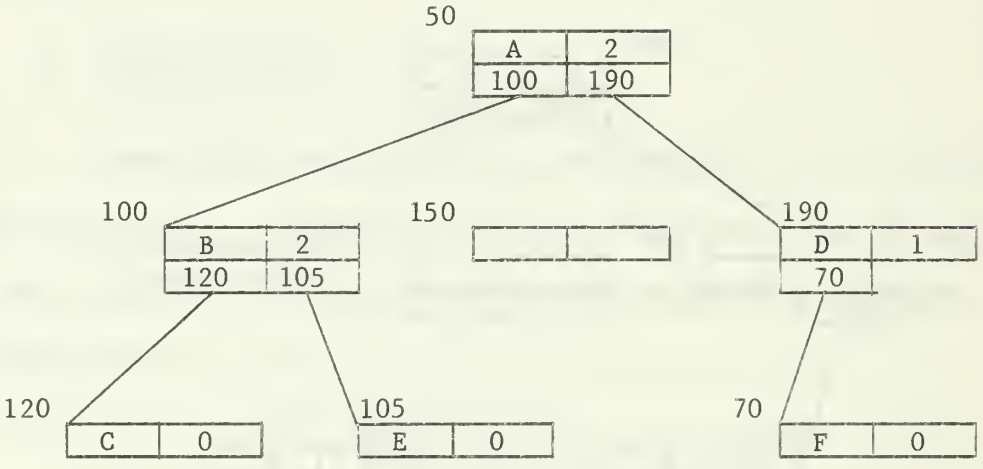


(b) After deleting node E

Figure 5. The Delete Operation



(a) Before adding node F



(b) After adding node F

Figure 6. The Add Operation

There is no practical limit on the number of trees which can be saved. Since SAVE is declared as a BASED structure using the REFER[1] option, it will be allocated as large as necessary to save the desired trees.

Refer to the program listing in Appendix B for explicit details on the implementation of the various functions of TREEPAK. Appendix A contains instructions on the interface requirements of the main program and user operation of the graphic display console. A sample application is described in the next section.

---

[1]The REFER option is used in the IBM implementation of PL/I to specify that the value of a variable outside a based structure is to be used to determine the length or bound of a variable within the structure upon allocation. In this example the value of TOP at the time of the allocation determines the upper bound for the dimension of ROOTS.

III. APPLICATION PROGRAM

A. DESCRIPTION OF THE WANG ALGORITHM

This section gives an example of a program which uses tree structures for representing strings of symbols in theorem proving. The algorithm is described in this section; a description of the implementation is given in the next section. The program listing is found in Appendix D.

The Wang Algorithm [9] is a mechanical method of determining whether or not a formula in the Propositional Calculus is a theorem. The language of the Propositional Calculus is defined by the Backus Naur Form statements in Table 1.

A sequent may be thought of as a statement which is true if and only if some formula in the string on the left side of the arrow (the antecedent) is false or some formula in the string on the right side of the arrow (the consequent) is true. That is, the conjunction of the formulas on the left implies the disjunction of the formulas on the right. The five connectives, NOT, AND, OR, EQUIV, IMPLIES, are given their usual logical interpretation.

```
⟨SEQUENT⟩  ::=  ⟨STRING⟩ ⟶ ⟨STRING⟩
⟨STRING⟩  ::=  ⟨EMPTY⟩ | ⟨STRING SET⟩
⟨STRING SET⟩  ::=  ⟨FORMULA⟩ | ⟨STRING SET⟩ , ⟨FORMULA⟩
⟨EMPTY⟩  ::=  {The null string of symbols}
⟨FORMULA⟩  ::=  ⟨PROPOSITIONAL LETTER⟩ | NOT( ⟨FORMULA⟩ ) |
          ( ⟨FORMULA⟩ ⟨BINARY CONNECTIVE⟩ ⟨FORMULA⟩ )
⟨BINARY CONNECTIVE⟩  ::=  AND | OR | IMPLIES | EQUIV
⟨PROPOSITIONAL LETTER⟩  ::=  {An alphabetic letter}
```

Table 1. Definition of the Language

In general, the algorithm proceeds as follows. The formula to be tested is first preceded with the arrow to form a sequent in which the antecedent is the null string and the consequent is the formula to be tested. The consequent is then reordered into a form of Polish notation. The connectives are eliminated from the sequent, beginning with the leftmost connective, according to ten rules of elimination. This process produces a finite set of sequents which contain strings of propositional letters on both sides of the arrow. The truth of each sequent produced is then tested according to the following initial rule:

1. If a,b are strings of propositional letters, then a⟶b is a theorem if and only if some propositional letter occurs on both sides of the arrow.

Ten rules of derivation are listed below. In the rules, a and b are always strings (possibly empty) of propositional letters; c, d, e, and f are strings. It will be noted that there are two rules for each logical connective, one introducing it on each side of the arrow. Using these rules, the usual proof procedure would begin with a finite number

2A. ⟶ NOT: If c,a⟶b,d then a⟶b,NOT(c),d.
2B. NOT ⟶ : If a,c⟶d,e then a,NOT(e),c⟶d.
3A. ⟶ AND: If a⟶b,c,d and a⟶b,e,d then a⟶b,(c AND e),d.
3B. AND ⟶ : If a,c,d,e⟶f then a,(c AND d),e⟶f.
4A. ⟶ OR: If a⟶b,c,d,e then a⟶b,(c OR d),e.
4B. OR ⟶ : If a,c,d⟶e and a,f,d⟶e then a,(c OR f),d⟶e.
5A. ⟶IMPLIES: If a,c⟶b,d,e then a⟶b,(c IMPLIES d),e.
5B. IMPLIES⟶: If a,c,d⟶e and a,d⟶e,f then a,(f IMPLIES d),d⟶e.
6A. ⟶EQUIV: If c,a⟶b,d,e and d,a⟶b,c,e then a⟶b,(c EQUIV d),e.
6B. EQUIV⟶: If c,d,a,e⟶f and a,e⟶f,c,d then a,(c EQUIV d),e⟶f.

Table 2. Rules of Derivation

of cases of Rule 1, and using the ten derivation rules, introduce the proper connectives to arrive at the sequent which represents the original theorem. In contrast to the usual proof procedure, the Wang Algorithm begins with the original theorem and applies the rules backwards to obtain connective-free sequents. The rules are designed so that given a sequent, the leftmost connective may be eliminated, thereby resulting in one or two premises which, taken together, are equivalent to the conclusion. This process is repeated until a finite set of sequents is reached which contain only propositional letters. The original sequent is a theorem if all of the sequents thus obtained are theorems when tested by the initial rule.

The following example illustrates the proof procedure. The first line is the formula to be tested, with the sequent arrow prefixed to it. Successive lines are the sequents generated by each elimination of a connective. Each line is preceded by the number of the rule which produced the sequent.

```
      ⟶((NOT(P) AND NOT(Q)) IMPLIES (P EQUIV Q))
5A   (NOT(P) AND NOT(Q))⟶(P EQUIV Q)
3B   NOT(P),NOT(Q)⟶(P EQUIV Q)
2B   NOT(Q)⟶(P EQUIV Q),P
2B   ⟶(P EQUIV Q),P,Q
6A   P⟶Q,P,Q
     VALID

6A   Q⟶P,P,Q
     VALID
```

The techniques used for implementing this algorithm are presented in the next section.

B. IMPLEMENTATION OF THE WANG ALGORITHM

The implementation of the Wang Algorithm requires a structure for representing the sequents and a set of routines for manipulating the structure according to the rules stated in the previous section. The sequent can be conveniently represented as a tree. The root of the tree represents the sequent arrow. The root then has two successors, the antecedent and the consequent. Each formula is represented as a subtree of the antecedent if it is on the left side of the arrow, or the consequent if it is on the right side. A binary connective is represented as a node with two branches: one for each of the formulas it connects. The unary connective, NOT, is a node with one successor, i.e., the formula it negates. The tree representation of the theorem shown in Figure 7 illustrates the structure.

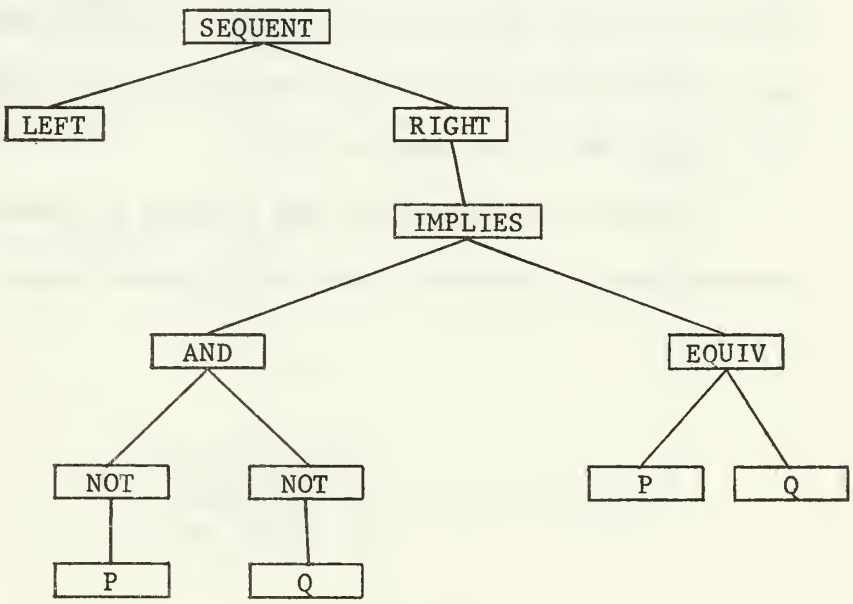


Figure 7. Representation of a Sequent

Certain changes in notation are made to facilitate the input of sequents in this tree form. The notation described in Section II.C for the description of trees in linear form is used. Thus the expression (P IMPLIES Q) becomes (IMPLIES(P,Q)). Using this notation, the statement of the theorem in the example is as follows:

(SEQUENT(LEFT,RIGHT(IMPLIES(AND(NOT(P),NOT(Q)),EQUIV(P,Q))))).

The theorem is input in this notation. The algorithm proceeds to eliminate the connectives one at a time. The routines eliminating the connectives are simply a set of tree manipulation routines which perform transformations on the current sequent until all connectives are removed. Five of the rules generate additional premises which must be tested for validity. These new sequents are stacked as they are generated. When the connectives are eliminated from the current sequent, it is tested and, if valid, the next sequent is obtained from the stack to become the current sequent. This process is repeated until all premises have been tested and the stack is empty. The program is then ready for a new theorem.

A general flow chart for the program is shown in Figure 8. The photographs in Appendix C are taken from a sample run of this program.

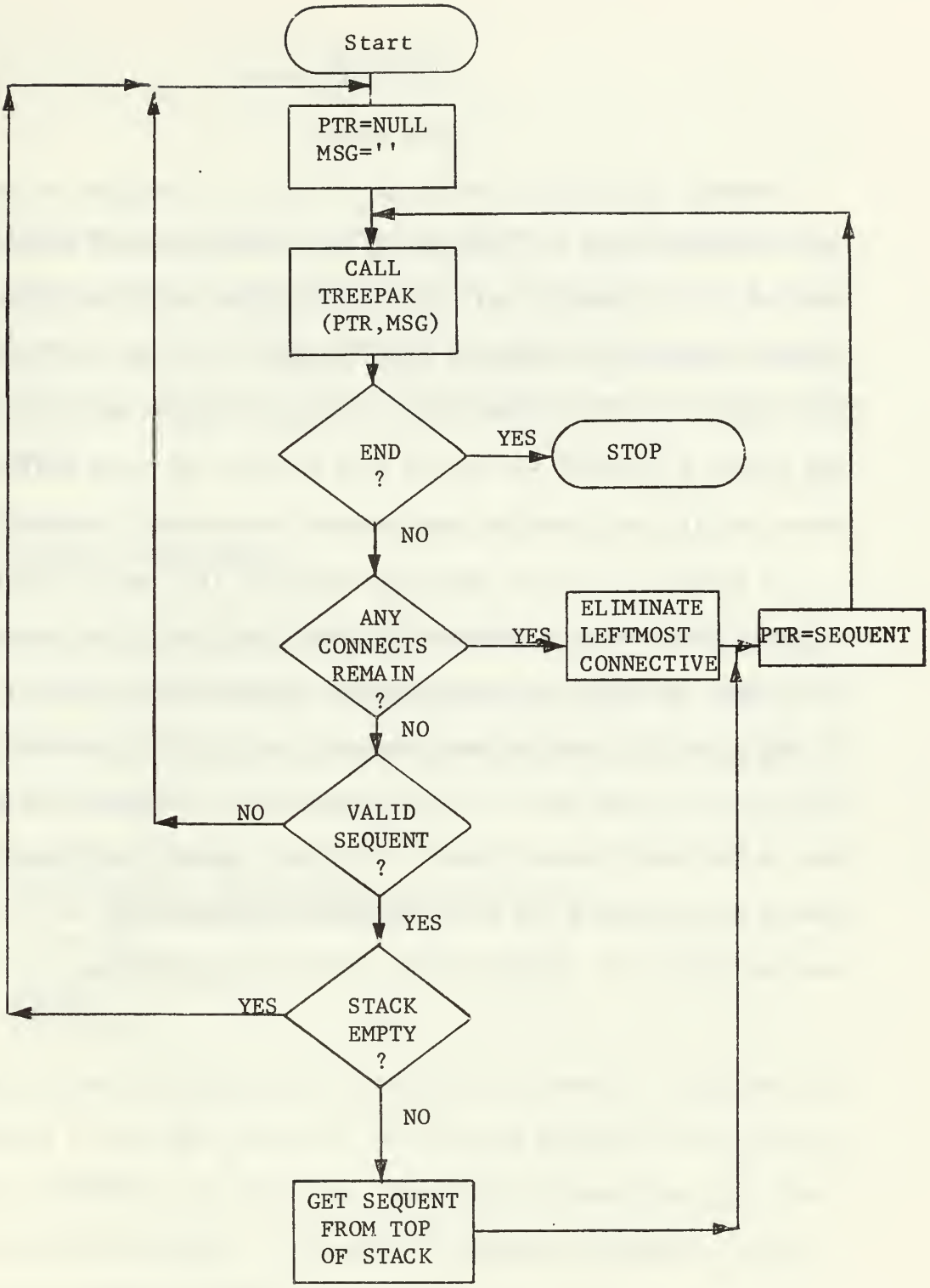


Figure 8. Wang Algorithm Flow Chart

IV. CONCLUSION

TREEPAK, described in Sections I and II, is designed to provide a user-oriented means for interacting with tree-type data structures. In Section III, an example was given of a program which uses tree structures to represent strings of logic symbols in an on-line theorem tester. This type of data structure occurs in many computer applications. TREEPAK can be a valuable aid to the user for many of these applications which fulfill the interface requirements described in Appendix A.

In Section II.D.2, it was shown that any list may be displayed as a tree. This technique provides a useful picture of the structure if the number of cycles is small compared to the total number of nodes. If the structure contains many cycles it may still be displayed as a tree, but the user may find that it is hard to interpret the picture due to the many repeated nodes. Thus, the primary usefulness of TREEPAK is in displaying and manipulating structures which are trees (contain no cycles) or "nearly" trees (contain few cycles).

APPENDIX A

USER'S GUIDE

The information contained in this appendix is not necessary for a general understanding of the system, nor is it intended to be a self-contained user's guide. It is provided for the potential user and contains details on how to access and operate TREEPAK not contained in the previous pages.

A. MAIN PROGRAM REQUIREMENTS

Main program requirements include a declaration of the arguments used in the call, a declaration of TREEPAK itself, and the call to TREEPAK. The following is an example of the minimum main program which can successfully call TREEPAK:

```
MAIN:PROCEDURE OPTIONS (MAIN);
     DCL P POINTER, MSG CHAR(74) VAR INIT(''),
           TREEPAK RETURNS(POINTER);
     P = NULL;
     P = TREEPAK(P,MSG);
     RETURN;
     END MAIN;
```

This program contains all the necessary statements. Note that in this example a tree does not exist in the main program at the time of the call to TREEPAK. In this case, the first argument must have the value of the NULL pointer. No message is passed to TREEPAK for display in this example, however, a character string of up to seventy-four characters may be passed as the second argument. When control returns to the main program a pointer to the root of the tree which was on display at the time of the return is placed in P.

If a tree is passed to TREEPAK, or if reference is made to the returned tree by the main program the structure of the nodes must be declared in the following format:

```
DCL TOP FIXED BIN,
    1 NODE BASED(P),
      2 NAME CHAR(10),
      2 DEGREE FIXED BIN,
      2 BRANCH(0 : TOP REFER (DEGREE));
```

When a tree is passed to TREEPAK for display a pointer to the same tree, which may have been altered by the user at the console, is returned to the main program.

It is permissable for the main program to access the graphic display unit directly without going through TREEPAK. The only restriction is that the data control block and attention handler used for the direct access must be closed prior to calling TREEPAK. TREEPAK closes its graphics data control block and attention handler before returning to the main program.

B. JOB CONTROL STATEMENTS

Job control statements required are those normally required for running a PL/I job and statements required for loading a copy of TREEPAK and WJS999. The following is an example of those statements required at Naval Postgraduate School:

```
//  Standard job card,TYPRUN=HOLD
//  EXEC  PL1LFCLG,REGION.GO=150K
//PL1L.SYSIN  DD  *

    User's main program

/*
//LKED.LIBA  DD  DSN=S0373.PLIGPAK,UNIT=2314,VOL=SER=LINDA,
          DISP=SHR
//LKED.SYSIN  DD  *
   LIBRARY  LIBA(TREEPAK,WJS999)
```

```
/*
//GO.DISPLAY  DD  UNIT=(2250-1)
//GO.SYSIN  DD  *
      Trees in list notation if desired (two cards per tree)
/*
```

The job card is a standard job card except that TYPRUN=HOLD is required for all jobs which use the graphic display unit at NPS. The data definition statement named LIBA describes a partitioned data set which contains two members, TREEPAK and WJS999, both in load module form. The data definition statement for the graphic display unit must be named DISPLAY.

C. USING THE PROGRAMMED FUNCTION KEYBOARD

Only those keys which are active at any particular time are lighted. For example, if no node has been identified, the keys which require an identified node are not lighted, if no trees have been saved, the key which retrieves a saved tree is not lighted. The following is a description of the function of each key.

PFK Key	Function
0	Read and display a tree from SYSIN. Two cards are read from the GO.SYSIN data set. The list description of a tree must appear on those cards. The list may begin any place on the cards, but there must not be any intervening blanks between the beginning and end of the list. Syntax errors cause an error message to the user and the card images are then displayed on the screen. At this time the user may correct the error found and create the tree by pressing PFK key 1. If no more cards are available to be read, a message to the user is generated.
1	Read and display a tree typed at console keyboard. This function causes the typing area of the screen to be read. The subsequent actions are the same as those for PFK 0 after reading the cards. The important point to note is that the list must be typed in before pressing the key.

PFK Key	Function
2	Save displayed tree in save area. The current tree is saved. If the tree is to be assigned a name, the name must be typed in before pressing the key. The name may be up to ten alphanumeric or special characters. If an identical name already appears in the save area, the old tree is replaced. The user is then informed of the number assigned to the tree by TREEPAK.
4	Display identified node structure. Execution of this function does not change the structure of the tree, it merely changes the display screen so that the identified node appears as the root. Only that portion of the current tree which is the substructure of the identified node appears.
5	Add structure to identified node. This key causes an asterisk to appear at all the possible positions in which a successor could be added to the identified node. A message instructs the user to type in the new successor, then touch the desired position with the light pen. Either a save number, a saved name or the list notation for a tree may be typed in. After the light pen is activated over one of the asterisks, the tree will be placed on the screen with the added structure in the position indicated. If, after pressing the key, the user does not wish to add a successor, the display may be reinitialized by using PFK key 18.
6	Delete identified node. The identified node and all that it dominates is deleted from the current tree. Pressing this key when the identified node is the root of the current tree will result in an error message. (See PFK 10 for deleting an entire tree from the save area).
7	Detach identified node and save. This operation is the same as for PFK key 6, except that the deleted subtree is saved in the save area. If the subtree is to be assigned a name, it must be typed in before pressing the key.
8	Save identified node. The subtree dominated by the identified node is saved, however, the structure of the current tree is not changed. The name assigned, if applicable, must be typed in before pressing the key.
10	Delete a tree from save area. The name or number of a tree in the save area must be typed in before pressing the key. That tree is then dropped from the save area, and will no longer be accessible by that name or number.

PFK Key	Function
11	Display a tree from save area. The name or number of a tree which has been saved must be typed in before pressing the key. This tree then becomes the current tree and is displayed on the screen. If the tree which was current at the time the key was pressed has not been saved, it will no longer be accessible.
16	Display names and numbers of trees in save area. A listing of the trees which have been saved is presented, including their names and numbers. When this display is on the screen, PFK 18 must be pressed before any further action is taken. All other functions are disabled during this time.
17	Display instructions. The underlined portions of the instructions printed here are displayed. PFK 18 must be pressed before continuing.
18	Terminate instruction display and/or reinitialize display. The display returns to the current tree if either PFK key 16 or 17 has been pressed. This key may also be used to reinitialize the display if PFK 5 is inadvertently pressed and the user does not wish to add any successor to the identified node.
31	Return to the main program. The display is terminated, the console interrupts disabled, and the data control block is closed before returning to the main program. A pointer is returned to the root of the tree which was passed as an argument when TREEPAK was called, or if the argument was the NULL pointer, a pointer to the root of the tree on display is returned. If the user wishes to return a character string to the main program, he may type it on the keyboard before returning. The character string will be placed in the character string which was used as an argument in the call.

```
TREEPAK:PROCEDURE(PTR,CHSTR) POINTER;
                TREEPAK PROGRAM LISTING


  /*          VARIABLES USED WITH GRAPHICS PACKAGE */

/* DATA AREA FOR ATTENTION ROUTINES */
        DCL   1   ATTNAREA,
                    2   GACB FLOAT(9),
                    2   PFKMSK  BIT(32) INIT((32) '1'B),
                    2   ATTNTYP  BIT(32),
                    2   EP,
                    2   DUM(9),
                    2   RSV  CHAR(1),
                    2   OVERLY  CHAR(1),
                    2   KEYNUM  CHAR(1),
                    2   TYPE  CHAR(1),
                    2   SENSE  BIT(16),
                    2   BUFPOS  BIT(16),
                    2   XPOS  CHAR(2),
                    2   YPOS  CHAR(2),
                    2   RSV3,
                    2   DSA(25),
                    2   REG14  INIT(0),
                    2   REG15  INIT(0),
                    2   REG13  INIT(0),
                    2   FLPT(8),
                    2   PRVPTR,
                    2   ADDLST(2);
/* PARAMETER TABLE SET UP BY XYLIM ROUTINE. SPECIFIES SCALING LIMITS */
        DCL 1 XYLIMZ,
                    2   X1  CHAR(2),
                    2   Y1  CHAR(2),
                    2   X2  CHAR(2),
                    2   Y2  CHAR(2),
                    2   U1  FIXED BIN(31),
                    2   U2  FIXED BIN(31),
                    2   V1  FIXED BIN(31),
                    2   V2  FIXED BIN(31);
```

```
/* PARAMETER TABLE FOR PASSING VALUES TO GCPRNT ROUTINE */

        DCL 1 GCPRNTZ,
                2  XYLIMZ,
                2  UC   FIXED BIN(31),
                2  VC   FIXED BIN(31),
                2  F   CHAR(1) INIT('B'),
                2  S   CHAR(1)  INIT('N'),
                2  CSIZE  CHAR(1)  INIT('B'),
                2  CMOD  CHAR(1)  INIT('P'),
                2  DATA,
                2  N   CHAR(2),
                2  START  CHAR(1)  INIT('U'),
                2  RESERVED  CHAR(2);

/* PARAMETER TABLE FOR PASSING VALUES TO GSVPLT ROUTINE */

        DCL 1 GSVPLTZ,
                2  XYLIMZ,
                2  UTAB,
                2  VTAB,
                2  F   CHAR(1)  INIT('B'),
                2  S   CHAR(1)  INIT('N'),
                2  CSIZE  CHAR(1)  INIT('B'),
                2  CMOD  CHAR(1)  INIT('P'),
                2  A   CHAR(1)  INIT('A'),
                2  I   CHAR(1)  INIT('W'),
                2  PTYPE  CHAR(1)  INIT('E'),
                2  S_G_OPT  CHAR(1)  INIT('B'),
                2  N   CHAR(2),
                2  GRAPHIC  CHAR(1) INIT('*'),
                2  E   CHAR(1)  INIT('O');

/* OACB CONTAINS POINTERS INTO THE GDOA */
        DCL OACBZ(12);

/* OCBP CONTAINS POINTERS TO OACB AND GDOA */
        DCL OCBPZ(2);

/* WORK AREA FOR OCBP ROUTINE */
        DCL WORKAREAZ(100);

/* BIT STRING FOR PFK LIGHTS CONTROL */
        DCL LITS BIT(32);

 /* NUMBERS WHICH DESCRIBE THE BEGINNING AND END OF EACH PICTURE  */
```

```
        DCL  (PICSTR,PICEND) FIXED BIN(31) EXTERNAL;

/* CONTAINS DCB FOR INPUT/OUTPUT FOR GRAPHIC DISPLAY UNIT */
        DCL DCB(30) FLOAT STATIC;

/* RETURN CODE FOR EACH ROUTINE WILL BE PLACED INTO THIS AREA  */
        DCL  RCODE  FIXED BIN(31)  EXTERNAL;

 /* NUMBER OF BYTES TO BE TRANSFERRED TO DISPLAY   */
        DCL COUNT FIXED BIN(31) INIT(4096);

 /* ADDRESS INTO WHICH FIRST BYTE OF DATA TO BE PLACED */
        DCL  BUFADD  FIXED BIN(31)  INIT(0);

/* DDNAME OF DD CARD TO BE PLACED INTO THIS AREA */
        DCL  DD  CHAR(8) INIT('DISPLAY');

/* BUFFER ADDRESS INTO WHICH THE CURSOR IS INSERTED */
        DCL CURADD FIXED BIN(31) INIT(22);

 /* DEFINES THE SCALING LIMITS.  NO SCALING USED */
        DCL LIMPARM(8) FIXED BIN(31) INIT(0,0,4095,4095);

/* GRAPHIC DATA OUTPUT AREA TRANSFERRED TO DISPLAY */

        DCL 1 BUFFER,
              2 GSRT BIT(16) INIT('0010101010000010'B),
              2 GEVM BIT(16) INIT('0010101000000010'B),
              2 XLOC1 BIT(16) INIT('0100000000000000'B),
              2 YLOC1 BIT(16) INIT('0000000000000000'B),
              2 GECP BIT(16) INIT('0010101001000100'B),
              2 MARK CHAR(4) INIT('    '),
              2 GEPM BIT(16) INIT('0010101000000000'B),
              2 XLOC2 BIT(16) INIT('0100000000000000'B),
              2 YLOC2 BIT(16) INIT('0000000011001000'B),
              2 GECF BIT(16) INIT('0010101001000000'B),
              2 LINE(2) CHAR(74) INIT((2)(74) ' '),
              2 GECP2 BIT(16) INIT('0010101001000100'B),
              2 PLINE CHAR(74) INIT((74)' '),
              2 GTRU BIT(32) INIT('00101010111111110000000000000000'B),
              2 GDOAZ(1992) CHAR(2);
        DCL NOP4 BIT(32) INIT('00101010110000000000000000000000'B);

          /* DECLARE FUNCTIONS */

        DCL (FATHER,ADD,DELETE,LOCATE) RETURNS (POINTER);
```

```
        DCL (FIND_NO,NUMBER,STO_ROOT,SAVE_NO) RETURNS (FIXED BIN);
/* VARIABLE USED TO SAVE POINTERS TO TREE ROOTS */
        DCL 1 SAVE BASED (PT),
              2 NUM FIXED BIN,
              2 ROOTS(0:TOP REFER (SAVE.NUM)),
                3 PTR POINTER,
                3 NAME CHAR(10);
/* DECLARE NODES OF THE TREE STRUCTURE */
        DCL PTR POINTER;
        DCL    1 NODE BASED (PTR),
                 2 NAME CHAR(10),
                 2 DEGREE BIN FIXED,
                 2 BRANCH(0:BOUND REFER(NODE.DEGREE)) POINTER;

/* VARIABLES USED BY MORE THAN ONE PROCEDURE */
        DCL (LEV,COL,BOUND,MAXLEV,(WIDTH,TIMES)(11)) FIXED BIN;
        DCL CARD CHAR(320) VARYING, WAIT_BIT BIT(1) INIT('0'B);
        DCL (XTAB,YTAB)(200) FIXED BIN(31), (NOC,TC) FIXED BIN(16),
            NODNO FIXED DECIMAL(2), NC CHAR(10);
        DCL CHSTR CHAR(74) VARYING, I FIXED BIN;

/* VARIABLES USED BY MORE THAN ONE ATTENTION TYPE */
        DCL ((RDBIT,ON) INIT('1'B), (SVBIT, IDBIT, TZBIT, OFF)
            INIT('0'B)) BIT(1), INBIT BIT(1) INIT('0'B);
        DCL (DIST,(LPMODE,TOP) INIT(0)) FIXED BIN;
        DCL (XSTRT,RNO,LNO) FIXED BIN(31), (NRT,IDENT,ROOT) POINTER;
        DCL (XLP,YLP,RTNO INIT(0)) FIXED BIN, (TDT,SPT) POINTER;

/* ROUTINE WHICH REMOVES LEADING BLANKS */
ENDBLNK:PROC;
        COL=1;
        DO WHILE (SUBSTR(CARD,COL,1)=' ');
           COL=COL+1;
           IF COL>LENGTH(CARD) THEN RETURN;
        END;
```

```
        RETURN;
        END ENDBLNK;

/* ROUTINE TO SAVE A TREE IN SAVEAREA */

STO_ROOT:PROC(Q) FIXED BIN;
        DCL Q POINTER,N FIXED BIN;
        N=SAVE_NO;
        IF N=0 THEN DO;
           TOP=SAVE.NUM+1;
           TZBIT='1'B;
           NRT=PT;
           ALLOCATE SAVE;
           SAVE=NRT->SAVE,BY NAME;
           SAVE.NUM=SAVE.NUM+1;
           SAVE.ROOTS(SAVE.NUM).PTR=Q;
           SAVE.ROOTS(SAVE.NUM).NAME=NC;
           FREE NRT->SAVE;
           BUFFER.PLINE=NC||' SAVED IN SAVENUMBER'||TOP;
           RETURN (TOP);
        END;
        SAVE.ROOTS(N).PTR=Q;
        BUFFER.PLINE=NC||' REPLACED IN SAVENUMBER'||N;
        RETURN (N);
        END STO_ROOT;

/* ROUTINE TO FIND THE SAVENUMBER OF A TREE */

SAVE_NO:PROC FIXED BIN;
        DCL (I,Z INIT(0)) FIXED BIN;
        CALL ENDBLNK;
        NC=' ';
        IF COL<=LENGTH(CARD) THEN DO;
           NC=CARD;
           IF UNSPEC(SUBSTR(NC,1,1))<UNSPEC('0') THEN GO TO COMP;
           COL=1;
           DO WHILE (SUBSTR(NC,COL,1)¬=' ');
              IF UNSPEC(SUBSTR(NC,COL,1))< UNSPEC('0') THEN DO;
                 NC=' ';
                 RETURN (Z);
              END;
              IF COL>=10 THEN GO TO RET;
              COL=COL+1;
           END;
    RET:   I=NC;
           RETURN (I);
```

```
   COMP:    DO I=1 TO SAVE.NUM;
               IF NC=SAVE.ROOTS(I).NAME THEN RETURN (I);
            END;
         END;
         RETURN (Z);
         END SAVE_NO;

/* ROUTINE WHICH TURNS ON THE PFK LIGHTS */

INIT:    PROC;
         LITS=RDBIT||ON||SVBIT||OFF||REPEAT(IDBIT,4)||OFF||TZBIT
            ||TZBIT||REPEAT(OFF,3)||ON||ON||
           ON||REPEAT(OFF,11)||ON;
         CALL WJS999('GCNTRL',DCB,'IND',LITS);
         CALL CHECK;
         RETURN;
         END INIT;

/* ROUTINE TO WRITE OUT DISPLAY AND INSERT CURSOR */

SHOW: PROC;
         CALL WJS999('GWRITE',DCB,COUNT,BUFFER,BUFADD);
         CALL CHECK;
         CALL WJS999('GCNTRL',DCB,'INS',CURADD);
         CALL CHECK;
         CALL WJS999('GCNTRL',DCB,'STR',BUFADD);
         CALL CHECK;
         RETURN;
         END SHOW;

/* THIS ROUTINE RETURNS A POINTER TO THE FATHER OF Q IN THE STRUCTURE
    POINTED TO BY R */

FATHER:PROCEDURE(R,Q) RECURSIVE POINTER;
         DCL (S,R,Q) POINTER,I BIN FIXED;
         IF R=Q THEN DO;
            S=NULL;
            RETURN(S);
         END;
         IF R->NODE.DEGREE<0 THEN GO TO RET;
         R->NODE.DEGREE=-R->NODE.DEGREE;
         DO I=1 TO -R->NODE.DEGREE;
            IF Q=R->NODE.BRANCH(I) THEN RETURN (R);
            S=FATHER((R->NODE.BRANCH(I)),Q);
            IF S¬=NULL THEN RETURN (S);
```

```
            END;
RET:        S=NULL;
            RETURN (S);
            END FATHER;

/* THIS ROUTINE COUNTS THE NUMBER OF NODES AT THE SAME LEVEL ON THE LEFT
 OF THE NODE POINTED TO BY Q IN THE STRUCTURE POINTED TO BY R */

FIND_NO:PROCEDURE(R,Q) FIXED BIN RECURSIVE;
            DCL (R,Q) POINTER,(I,N) FIXED BIN;
            LEV=LEV+1;
            TIMES(LEV)=TIMES(LEV)+1;
            IF R=Q THEN RETURN (TIMES(LEV));
            IF LEV>=10|TIMES(LEV+1)>=20 THEN GO TO RET;
            IF R->NODE.DEGREE<0 THEN GO TO RET;
            R->NODE.DEGREE=-R->NODE.DEGREE;
            DO I=1 TO -R->NODE.DEGREE;
               N=FIND_NO((R->NODE.BRANCH(I)),Q);
               IF N¬=0 THEN RETURN(N);
            END;
      RET:  LEV=LEV-1;
            RETURN(0);
            END;


/* THIS ROUTINE ADDS STRUCTURE POINTED TO BY R TO NODE POINTED TO BY Q
IN POSITION N.  RETURNS POINTER TO NODE WHICH REPLACED Q */

ADD:        PROCEDURE(Q,R,N) POINTER;
            DCL (Q,R,S) POINTER, (I,N) FIXED BIN;
            BOUND=Q->NODE.DEGREE+1;
            ALLOCATE NODE;
            NODE.NAME=Q->NODE.NAME;
            DO I=1 TO N-1;
               NODE.BRANCH(I)=Q->NODE.BRANCH(I);
            END;
            NODE.BRANCH(N)=R;
            DO I=N+1 TO NODE.DEGREE;
               NODE.BRANCH(I)=Q->NODE.BRANCH(I-1);
            END;
            IF Q=SPT THEN SPT=PTR;
            FREE Q->NODE;
            RETURN (PTR);
            END ADD;

/* THIS ROUTINE DELETES THE NTH BRANCH FROM THE NODE POINTED TO BY Q.
```

```
  RETURNS A POINTER TO THE NODE WHICH REPLACED Q. */

DELETE: PROCEDURE(Q,N) POINTER;
        DCL Q POINTER, (N,I) FIXED BIN;
        BOUND=Q->NODE.DEGREE-1;
        ALLOCATE NODE;
        NODE.NAME=Q->NODE.NAME;
        DO I=1 TO N-1;
           NODE.BRANCH(I)=Q->NODE.BRANCH(I);
        END;
        DO I=N+1 TO Q->NODE.DEGREE;
           NODE.BRANCH(I-1)=Q->NODE.BRANCH(I);
        END;
        IF Q=SPT THEN SPT=PTR;
        FREE Q-> NODE;
        RETURN (PTR);
        END DELETE;

/* ROUTINE TO READ A LIST INPUT FROM THE CRT */

READ_CRT: PROCEDURE;
        CALL WJS999('GREAD',DCB,COUNT,BUFFER,BUFADD);
        CALL CHECK;
        CARD=BUFFER.LINE(1)||BUFFER.LINE(2);
        RETURN;
        END READ_CRT;

/* GIVEN THAT Q IS A BRANCH OF R THIS ROUTINE RETURNS THE BRNCH NUM*/

NUMBER: PROCEDURE(R,Q) BIN FIXED;
        DCL (R,Q) POINTER, I BIN FIXED;
        DO I=1 TO R->NODE.DEGREE;
           IF Q=R->NODE.BRANCH(I) THEN RETURN (I);
        END;
        RETURN (0);
        END NUMBER;

/* ROUTINE TO LOCATE THE NODE POINTED OUT BY THE LIGHT PEN */

LOCATE: PROC(R,L,N) POINTER RECURSIVE;
        DCL (R,S) POINTER,(L,N,I) FIXED BIN(31);
        LEV=LEV+1;
        TIMES(LEV)=TIMES(LEV)+1;
        IF LEV=L THEN IF TIMES(LEV)=N THEN RETURN (R);
        IF R->NODE.DEGREE<0 THEN GO TO RET;
        R->NODE.DEGREE=-R->NODE.DEGREE;
```

```
            DO I=1 TO -R->NODE.DEGREE;
               S=LOCATE(R->NODE.BRANCH(I),L,N);
               IF S¬=NULL THEN RETURN (S);
            END;
      RET:  LEV=LEV-1;
            S=NULL;
            RETURN (S);
            END LOCATE;

/*  ROUTINE TO RESET THE NODE.DEGREES WHICH WERE CHANGED IN
 TRAVERSING THE TREE  */

RESET:   PROC(B) RECURSIVE;
            DCL B POINTER,I BIN FIXED;
            IF B=NULL THEN RETURN;
            IF B->NODE.DEGREE>=0 THEN RETURN;
            B->NODE.DEGREE=-B->NODE.DEGREE;
            DO I=1 TO B-> NODE.DEGREE;
               CALL RESET(B->NODE.BRANCH(I));
            END;
            RETURN;
            END RESET;


 /*  ROUTINE TO READ A CARD FROM THE INPUT STREAM */

READ_CARD:PROC;
            ON ENDFILE(SYSIN) BEGIN;
            BUFFER.PLINE=' ALL CARDS HAVE BEEN READ';
            CALL SHOW;
            RDBIT='0'B;
            CALL INIT;
            GO TO RET;
            END;
            GET EDIT (CARD) (COLUMN(1),A(160));
 RET:       RETURN;
            END READ_CARD;

 /* ROUTINE TO BUILD THE TREE STRUCTURE FROM THE STRING    */

PASS1:   PROC;

 /* TEMPORARY LINEAR LIST TO HOLD NODE INFORMATION FOR TREE */
            DCL P POINTER;
            DCL     1 TEMP BASED (P),
                      2 NAME CHAR(10),
```

```
                    2 DEGREE BIN FIXED,
                    2 LEVEL  BIN FIXED,
                    2 NEXT POINTER;
        DCL (HEAD,LAST,(LSTNODE,LSTEMP) CONTROLLED) POINTER;
        DCL (I,DIFF) BIN FIXED, SYM RETURNS (CHAR(10));

/* ROUTINE TO GET THE NODE NAME FROM A CHARACTER STRING */

SYM: PROCEDURE CHAR(10);
        DCL T BIN FIXED, C CHAR(1);
        T=COL;
        DO WHILE (UNSPEC(SUBSTR(CARD,COL,1))>=UNSPEC('A'));
           COL=COL+1;
        END;
        IF T=COL THEN RETURN ('');
        C=SUBSTR(CARD,COL,1);
        IF LEV=1 THEN IF C=',' THEN RETURN ('');
        IF C¬='(' THEN
            IF C¬=',' THEN
                IF C¬=')' THEN RETURN ('');
        RETURN (SUBSTR(CARD,T,COL-T));
        END SYM;

        LEV=0;
        MAXLEV=0;
        CALL ENDBLNK;
        IF SUBSTR(CARD,COL,1)¬='(' THEN GO TO ERRMSG;
        COL=COL+1;
        LEV=LEV+1;
        ALLOCATE TEMP;
        TEMP.DEGREE=0;
        HEAD=P;
ASSIGN: LAST=P;
        TEMP.NAME=SYM;
        IF TEMP.NAME='' THEN GO TO ERRMSG;
        TEMP.LEVEL=LEV;
        IF SUBSTR(CARD,COL,1)='(' THEN DO;
           TEMP.DEGREE=TEMP.DEGREE+1;
           ALLOCATE LSTNODE;
           LSTNODE=P;
           LEV=LEV+1;
           COL=COL+1;
           ALLOCATE TEMP;
           TEMP.DEGREE=0;
           LAST->TEMP.NEXT=P;
           GO TO ASSIGN;
```

```
          END;
COMMA:    IF SUBSTR(CARD,COL,1)=',' THEN DO;
             LSTNODE->TEMP.DEGREE=LSTNODE->TEMP.DEGREE+1;
             COL=COL+1;
             ALLOCATE TEMP;
             TEMP.DEGREE=0;
             LAST->TEMP.NEXT=P;
             GO TO ASSIGN;
          END;
          IF SUBSTR(CARD,COL,1)=')' THEN DO;
             LEV=LEV-1;
             COL=COL+1;
             FREE LSTNODE;
          IF LEV=1 THEN IF SUBSTR(CARD,COL,1)¬=')' THEN GO TO ERRMSG;
             IF LEV=0 THEN GO TO PROCEED;
             GO TO COMMA;
          END;
 ERRMSG:BUFFER.PLINE='SYNTAX ERROR BEGINNING IN COLUMN'||COL;
          BUFFER.LINE(1)=SUBSTR(CARD,1,74);
          BUFFER.LINE(2)=SUBSTR(CARD,75);
          CALL SHOW;
          ROOT=NULL;
          RETURN;
PROCEED:LAST->TEMP.NEXT=NULL;
          P,LAST=HEAD;
          BOUND=TEMP.DEGREE;
          ALLOCATE NODE;
          ROOT=PTR;
          NODE=TEMP,BY NAME;
          P=TEMP.NEXT;
          DO WHILE (P¬=NULL);
             IF TEMP.LEVEL>LAST->TEMP.LEVEL THEN DO;
                ALLOCATE LSTNODE,LSTEMP;
                LSTNODE=PTR;
                LSTEMP=LAST;
                LSTEMP->TEMP.DEGREE=1;
             END;
             IF TEMP.LEVEL<LAST->TEMP.LEVEL THEN DO;
                DIFF=LAST->TEMP.LEVEL-TEMP.LEVEL;
                DO I=1 TO DIFF;
                   FREE LSTNODE,LSTEMP;
                END;
             END;
             BOUND=TEMP.DEGREE;
             ALLOCATE NODE;
             NODE=TEMP,BY NAME;
```

```
            LSTNODE->NODE.BRANCH(LSTEMP->TEMP.DEGREE)=PTR;
            LSTEMP->TEMP.DEGREE=LSTEMP->TEMP.DEGREE+1;
            LAST=P;
            P=TEMP.NEXT;
         END;
         PTR=ROOT;
         P=HEAD;
         DO WHILE (P¬=NULL);
            HEAD=TEMP.NEXT;
            FREE TEMP;
            P=HEAD;
         END;
         RETURN;
         END PASS1;

/* ROUTINE TO CHECK THE RETURN CODE  */

CHECK:PROC;
         IF RCODE¬=0 THEN DO;
             PUT DATA (RCODE) SKIP;
             CALL IHEDUMP;
         END;
         RETURN;
         END CHECK;

 /* ROUTINE TO DISPLAY THE TREE ON THE CRT  */

DSPLY:PROC;
         DCL PC CHAR(1) INIT('%'), I BIN FIXED;
 /* ROUTINE TO CALCULATE THE COORDINATES OF EACH NODE FOR DISPLAY */

GET_COORDS:PROC(Q) RECURSIVE;
         DCL PLUS CHAR(1) INIT('+'), INFO CHAR(10) VARYING;
         DCL (XCOORD,YCOORD,I) FIXED BIN(31), Q POINTER;
         DCL DNAM CHAR(13),CH5 CHAR(5),CH2 CHAR(2);
         NODNO=NODNO+1;
         IF NODNO>50 THEN DO;
            BUFFER.PLINE='BUFFER CAPACITY EXCEEDED. THE TREE SHOWN '
            ||'MAY NOT BE COMPLETE';
            RETURN;
         END;
         INFO=Q->NODE.NAME;
         COL=10;
         DO WHILE (SUBSTR(INFO,COL,1)=' ');
            COL=COL-1;
```

```
      END;
      INFO=SUBSTR(INFO,1,COL);
      CARD=CARD||INFO;
      LEV=LEV+1;
      TIMES(LEV)=TIMES(LEV)+1;
      XCOORD=(4096*TIMES(LEV))/(WIDTH(LEV)+1);
      YCOORD=4096-((4096*LEV)/(MAXLEV+1));
      NOC=2;
      UNSPEC(GCPRNTZ.N)=NOC;
      CH5=NODNO;
      CH2=SUBSTR(CH5,4);
      GCPRNTZ.VC=YCOORD+50;
      GCPRNTZ.UC=XCOORD;
      CALL WJS999('GCPRNT',OCBPZ,XYLIMZ,GCPRNTZ,CH2);
      CALL CHECK;
      DNAM=CH2||' '||INFO;
      NOC=13;
      UNSPEC(GCPRNTZ.N)=NOC;
      IF NODNO<=40 THEN DO;
         GCPRNTZ.VC=3990-NODNO*80;
         GCPRNTZ.UC=0;
      END; ELSE DO;
         GCPRNTZ.VC=3990-(NODNO-40)*80;
         GCPRNTZ.UC=3268;
      END;
      CALL WJS999('GCPRNT',OCBPZ,XYLIMZ,GCPRNTZ,DNAM);
      CALL CHECK;
      TC=TC+1;
      XTAB(TC)=XCOORD;
      YTAB(TC)=YCOORD;
      IF LEV>=10|TIMES(LEV+1)>=20 THEN DO;
         BUFFER.PLINE=' + INDICATES UNDERLYING STRUCTURE NOT SHOWN '
         ||'DUE TO SCREEN SPACE';
         NOC=1;
         UNSPEC(GCPRNTZ.N)=NOC;
         GCPRNTZ.UC=XCOORD-40;
         GCPRNTZ.VC=YCOORD+50;
         CALL WJS999('GCPRNT',OCBPZ,XYLIMZ,GCPRNTZ,PLUS);
         CALL CHECK;
         CARD=CARD||PLUS;
         GO TO RET;
      END;
   IF Q->NODE.DEGREE>0 THEN DO;
         CARD=CARD||PC;
         NOC=1;
         UNSPEC(GCPRNTZ.N)=NOC;
```

```
            GCPRNTZ.UC=XCOORD-40;
            GCPRNTZ.VC=YCOORD+50;
            CALL WJS999('GCPRNT',OCBPZ,XYLIMZ,GCPRNTZ,PC);
            CALL CHECK;
            BUFFER.PLINE='% INDICATES CYCLE IN TREE';
            GO TO RET;
         END;
         Q->NODE.DEGREE=-Q->NODE.DEGREE;
         DO I=1 TO Q->NODE.DEGREE;
            IF I=1 THEN CARD=CARD||'('; ELSE CARD=CARD||',';
            CALL GET_COORDS(Q->NODE.BRANCH(I));
            IF I=Q->NODE.DEGREE THEN CARD=CARD||')';
            TC=TC+1;
            XTAB(TC)=XCOORD;
            YTAB(TC)=YCOORD;
         END;
RET:     LEV=LEV-1;
         RETURN;
         END GET_COORDS;
```



```
 /*  ROUTINE TO GET INFO USED BY GET_COORDS ROUTINE */

GET_LEVEL:PROCEDURE(F) RECURSIVE;
         DCL F POINTER, I FIXED BIN;
         NODNO=NODNO+1;
         IF NODNO>50 THEN RETURN;
         LEV=LEV+1;
         IF LEV>MAXLEV THEN MAXLEV=LEV;
         WIDTH(LEV)=WIDTH(LEV)+1;
         IF LEV>=10|WIDTH(LEV+1)>=20 THEN GO TO RET;
         IF F->NODE.DEGREE<0 THEN GO TO RET;
         F->NODE.DEGREE=-F->NODE.DEGREE;
         DO I=1 TO -F->NODE.DEGREE;
            CALL GET_LEVEL(F->NODE.BRANCH(I));
         END;
RET:     LEV=LEV-1;
         RETURN;
         END GET_LEVEL;


         CALL WJS999('OACB',GDOAZ,OACBZ);
         CALL CHECK;
```

```
/* THIS ENTRY USED WHEN OACB SHOULD NOT BE INITIALIZED */

LDSPLY:ENTRY;
        LEV=0;
        WIDTH=0;
        MAXLEV=0;
        NODNO=0;
        CALL GET_LEVEL(ROOT);
        LEV=0;
        TIMES=0;
        TC=0;
        NODNO=0;
        CARD='(';
        CALL GET_COORDS(ROOT);
        CARD=CARD||')';
        NOC=LENGTH(CARD);
        UNSPEC(GCPRNTZ.N)=NOC;
        GCPRNTZ.VC=4050;
        GCPRNTZ.UC=0;
        UNSPEC(GSVPLTZ.N)=TC;
        CALL WJS999('GSVPLT',OCBPZ,XYLIMZ,GSVPLTZ,XTAB,YTAB);
        CALL CHECK;
        CALL WJS999('GCPRNT',OCBPZ,XYLIMZ,GCPRNTZ,CARD);
        CALL CHECK;
        CALL WJS999('GSTOR',OCBPZ,'END');
        CALL CHECK;
        CALL SHOW;
        RETURN;
        END DSPLY;

 /* ATTENTION HANDLER FOR PFK BUTTONS  */

        ON CONDITION (PFK  ) BEGIN;
        DCL ASTER CHAR(1) INIT('*'),BRNO FIXED BIN, SCARD CHAR(148);
        DCL TAG(0:31) LABEL,(FTHR,GFTHR,P) POINTER, (I,J) FIXED BIN;
        DCL 1  INSTS,
              2 GSRT BIT(16) INIT('0010101010000010'B),
              2 GEVM BIT(16) INIT('0010101000000010'B),
              2 XLOC BIT(16) INIT('0100000000000000'B),
              2 YLOC BIT(16) INIT('0000111000000000'B),
              2 GEPL BIT(16) INIT('0010101001000101'B),
              2 HEAD CHAR(49) INIT('                    INSTRUCTIONS'),
              2 NILL BIT(8) INIT('00000000'B),
              2 GEPB BIT(16) INIT('0010101001000100'B),
```

```
            2 BLNK(2) CHAR(74) INIT((2)(74) ' '),
            2 L(25) CHAR(74),
            2 GTRU BIT(32) INIT('00101010111111110000000000000000'B);
        LPMODE=0;
        BUFFER.PLINE=' ';
        IF INBIT='1'B THEN IF UNSPEC(KEYNUM)¬='00010010'B THEN GO TO
            PFKEND;
        BUFFER.LINE=' ';
        GO TO TAG(UNSPEC(ATTNAREA.KEYNUM));

/* READ AND DISPLAY TREE FROM SYSIN */

  TAG(0):CALL READ_CARD;
        IF RDBIT='0'B THEN GO TO  PFKEND;
        P=ROOT;
        CALL PASS1;
        IF ROOT=NULL THEN DO;
           ROOT=P;
           GO TO PFKEND;
        END;
        TDT=ROOT;
        BUFFER.MARK='    ';
        IDENT=NULL;
        IDBIT='0'B;
        SVBIT='1'B;
        BUFFER.GTRU=NOP4;
        CALL INIT;
        CALL DSPLY;
        RTNO=0;
        GO TO PFKEND;

/* READ AND DISPLAY TREE FROM CONSOLE */

  TAG(1):CALL READ_CRT;
        BUFFER.LINE=' ';
        IF CARD=' ' THEN DO;
           BUFFER.PLINE=' TYPE IN LIST BEFORE PRESSING PFK KEY 1';
           CALL SHOW;
           GO TO PFKEND;
        END;
        BUFFER.PLINE=' ';
        P=ROOT;
        CALL PASS1;
        IF ROOT=NULL THEN DO;
           ROOT=P;
           GO TO PFKEND;
```

```
        END;
        TDT=ROOT;
        SVBIT='1'B;
        IDBIT='0'B;
        BUFFER.MARK='     ';
        IDENT=NULL;
        BUFFER.GTRU=NOP4;
        CALL INIT;
        CALL DSPLY;
        RTNO=0;
        GO TO PFKEND;

/* SAVE DISPLAYED TREE */

 TAG(2):CALL READ_CRT;
        BUFFER.LINE=' ';
        RTNO=STO_ROOT(ROOT);
        SVBIT='0'B;
        CALL INIT;
        CALL SHOW;
        GO TO PFKEND;

/* DISPLAY STRUCTURE OF IDENT */

 TAG(4):IF IDENT=NULL THEN GO TO PFKEND;
        ROOT=IDENT;
        SVBIT='1'B;
        IDBIT='0'B;
        BUFFER.MARK=' ';
        IDENT=NULL;
        CALL INIT;
        CALL DSPLY;
        GO TO PFKEND;

/* ADD STRUCTURE TO IDENT FROM CONSOLE */

 TAG(5):IF IDENT=NULL THEN GO TO PFKEND;
        LPMODE=1;
        BUFFER.PLINE='TYPE IN TREE FOR ADDITION, THEN LIGHT PEN DESIRE'
        ||'D POSITION';
PUTSPOTS:CALL WJS999('0ACB',GDOAZ,0ACBZ);
        CALL CHECK;
        DIST= 4096/(WIDTH(LNO+1)+1);
        NOC=1;
        UNSPEC(GCPRNTZ.N)=NOC;
        IF IDENT->NODE.DEGREE=0 THEN DO;
```

```
            XSTRT=XLP;
            GCPRNTZ.UC=XLP;
            GCPRNTZ.VC=YLP-(4096/(MAXLEV+1))/2;
            CALL WJS999('GCPRNT',OCBPZ,XYLIMZ,GCPRNTZ,ASTER);
            CALL CHECK;
            CALL LDSPLY;
            GO TO PFKEND;
         END;
         LEV=0;
         TIMES=0;
         BRNO=FIND_NO(ROOT,(IDENT->NODE.BRANCH(1)));
         CALL RESET(ROOT);
         IF BRNO=0 THEN GO TO SYSERR;
         XSTRT=(4096*BRNO)/(WIDTH(LNO+1)+1)-DIST/2;
         GCPRNTZ.VC=YLP-(4096/(MAXLEV+1));
         DO I=1 TO IDENT->NODE.DEGREE+1;
            GCPRNTZ.UC=XSTRT +(I-1)*DIST;
            CALL WJS999('GCPRNT',OCBPZ,XYLIMZ,GCPRNTZ,ASTER);
            CALL CHECK;
         END;
         CALL LDSPLY;
         GO TO PFKEND;

/* DELETE IDENT FROM STRUCTURE */

 TAG(6):IF IDENT=NULL THEN GO TO PFKEND;
         IF ROOT=IDENT THEN DO;
            BUFFER.PLINE='ILLEGAL OPERATION. IDENTIFIED NODE IS ROOT';
            CALL SHOW;
            GO TO PFKEND;
         END;
         FTHR=FATHER(TDT,IDENT);
         CALL RESET(TDT);
         IF FTHR=NULL THEN GO TO SYSERR;
         I=NUMBER(FTHR,IDENT);
         IF I=0 THEN GO TO SYSERR;
         IF FTHR=TDT THEN DO;
            TDT=DELETE(FTHR,I);
            ROOT=TDT;
            IF RTNO¬=0 THEN SAVE.ROOTS(RTNO).PTR=TDT;
            IDBIT='0'B;
            BUFFER.MARK=' ';
            IDENT=NULL;
            CALL INIT;
            CALL DSPLY;
            GO TO PFKEND;
```

```
            END;
         GFTHR=FATHER(TDT,FTHR);
         CALL RESET(TDT);
         IF GFTHR=NULL THEN GO TO SYSERR;
         J=NUMBER(GFTHR,FTHR);
         IF J=0 THEN GO TO SYSERR;
         GFTHR->NODE.BRANCH(J)=DELETE(FTHR,I);
         IDBIT='0'B;
         BUFFER.MARK=' ';
         IDENT=NULL;
         CALL INIT;
         CALL DSPLY;
         GO TO PFKEND;

/* DETACH IDENT AND SAVE */

 TAG(7):IF IDENT=NULL THEN GO TO PFKEND;
         IF ROOT=IDENT THEN GO TO TAG(6);
         IF TDT=IDENT THEN GO TO TAG(8);
         CALL READ_CRT;
         BUFFER.LINE=' ';
         SCARD=CARD;
         FTHR=FATHER(TDT,IDENT);
         CALL RESET(TDT);
         IF FTHR=NULL THEN GO TO SYSERR;
         I=NUMBER(FTHR,IDENT);
         IF I=0 THEN GO TO SYSERR;
         IF FTHR=TDT THEN DO;
            TDT=DELETE(FTHR,I);
            ROOT=TDT;
            IF RTNO¬=0 THEN SAVE.ROOTS(RTNO).PTR=TDT;
            IDBIT='0'B;
            BUFFER.MARK=' ';
            CALL INIT;
            CALL DSPLY;
            GO TO TAG(8);
         END;
         GFTHR=FATHER(TDT,FTHR);
         CALL RESET(TDT);
         IF GFTHR=NULL THEN GO TO SYSERR;
         J=NUMBER(GFTHR,FTHR);
         IF J=0 THEN GO TO SYSERR;
         GFTHR->NODE.BRANCH(J)=DELETE(FTHR,I);
         IDBIT='0'B;
         BUFFER.MARK=' ';
         CALL DSPLY;
```

```
/* SAVE IDENT */

 TAG(8):IF IDENT =NULL THEN GO TO PFKEND;
        IF IDBIT='0'B THEN CARD=SCARD;
        ELSE DO;
           CALL READ_CRT;
           IDBIT='0'B;
        END;
        TZBIT='1'B;
        BUFFER.MARK=' ';
        BUFFER.LINE=' ';
        I=STO_ROOT(IDENT);
        CALL SHOW;
        CALL INIT;
        IDENT=NULL;
        GO TO PFKEND;

/* DELETE TREE FROM SAVEAREA */

 TAG(10):CALL READ_CRT;
        BUFFER.LINE=' ';
        I=SAVE_NO;
        IF I=0|I>SAVE.NUM THEN GO TO NISA;
        IF I=RTNO THEN RTNO=0;
        NRT=PT;
        TOP=SAVE.NUM-1;
        IF TOP=0 THEN DO;
           TZBIT='0'B;
          CALL INIT;
        END;
        ALLOCATE SAVE;
        DO J=1 TO I-1;
           SAVE.ROOTS(J)=NRT->SAVE.ROOTS(J),BY NAME;
        END;
        DO J=I+1 TO NRT->SAVE.NUM;
           SAVE.ROOTS(J-1)=NRT->SAVE.ROOTS(J),BY NAME;
        END;
        BUFFER.PLINE='TREE IN SAVENUMBER'||I||' DELETED: '
           ||NRT->SAVE.ROOTS(I).NAME;
        CALL SHOW;
        FREE NRT->SAVE;
        GO TO PFKEND;

/* DISPLAY TREE FROM SAVEAREA */
```

```
 TAG(11):CALL READ_CRT;
         BUFFER.LINE=' ';
         I=SAVE_NO;
         IF I=0|I>SAVE.NUM THEN DO;
  NISA:     BUFFER.PLINE='TREE NOT FOUND IN SAVEAREA';
            CALL SHOW;
            GO TO PFKEND;
         END;
         ROOT=SAVE.ROOTS(I).PTR;
         TDT=ROOT;
         RTNO=I;
            IDENT=NULL;
         IDBIT='0'B;
         BUFFER.MARK=' ';
         CALL INIT;
         CALL DSPLY;
         GO TO PFKEND;

/* NUMBER OF TREES IN SAVEAREA */

 TAG(16):INSTS.HEAD='                    SAVE AREA';
         IF SAVE.NUM<=24 THEN J=SAVE.NUM; ELSE J=24;
         DO I=1 TO J;
            L(I)=I||'  '||SAVE.ROOTS(I).NAME;
         END;
         DO I=J+1 TO 24;
            L(I)=' ';
         END;
         L(25)='         PRESS PFK 18 BEFORE CONTINUING';
         CALL WJS999('GWRITE',DCB,COUNT,INSTS,BUFADD);
         CALL CHECK;
         LITS='00000000000000001'B;
         CALL WJS999('GCNTRL',DCB,'IND',LITS);
         CALL CHECK;
         INBIT='1'B;
         GO TO PFKEND;

/* DISPLAY INSTRUCTIONS */

TAG(17):L(1)='    TO IDENTIFY A NODE, TOUCH IT WITH THE LIGHT PEN.  '
           ||'THIS MUST BE DONE';
         L(2)=' BEFORE PRESSING ANY PFK KEY WHICH REQUIRES AN '
           ||'IDENTIFIED NODE';
         L(3)=' ';
         L(4)=' ';
```

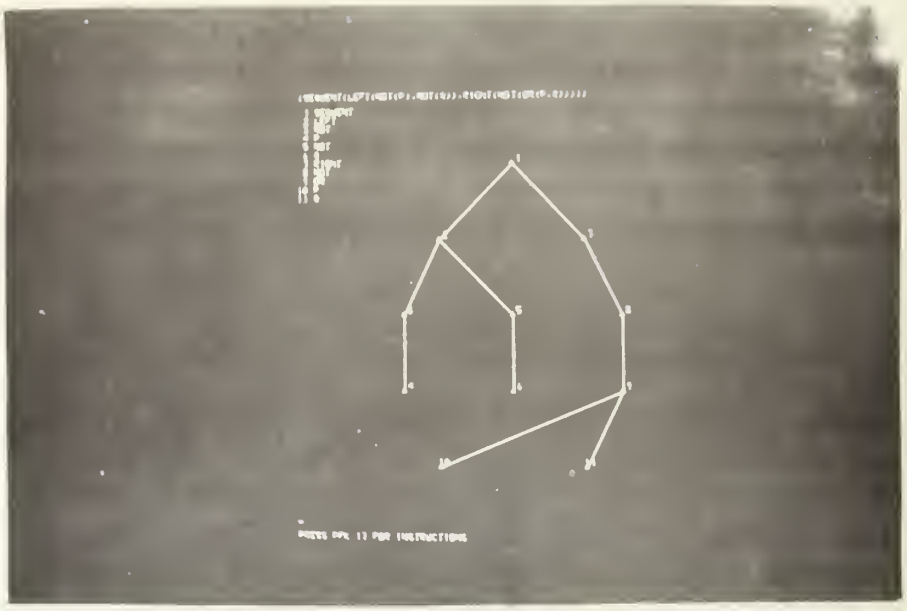
```
            L(5)='           PFK           OPERATION';
            L(6)='             0  READ AND DISPLAY TREE FROM SYSIN';
            L(7)='             1  READ AND DISPLAY TREE TYPED AT CONSOLE '
              ||'KEYBOARD';
            L(8)='             2  SAVE DISPLAYED TREE IN SAVEAREA';
            L(9)='             4  DISPLAY IDENTIFIED NODE STRUCTURE';
            L(10)='             5 *ADD STRUCTURE TO IDENTIFIED NODE';
            L(11)='             6  DELETE IDENTIFIED NODE';
            L(12)='             7  DETACH IDENTIFIED NODE AND SAVE';
            L(13)='             8  SAVE IDENTIFIED NODE';
            L(14)='            10  DELETE A TREE FROM SAVEAREA';
            L(15)='            11  DISPLAY A TREE FROM SAVEAREA';
            L(16)='            16  DISPLAY NUMBERS AND NAMES OF TREES IN'
              ||' SAVEAREA';
            L(17)='            17  DISPLAY INSTRUCTIONS';
            L(18)='            18  TERMINATE INSTRUCTION DISPLAY AND/OR '
              ||'REINITIALIZE DISPLAY';
            L(19)='            31  RETURN TO MAIN PROGRAM';
            L(20)=' ';
            L(21)=' ';
            L(22)='                * FURTHER INSTRUCTIONS AFTER PRESSING '
              ||'PFK KEY';
            L(23)=' ';
            L(24)=' ';
            L(25)='            PRESS PFK KEY 18 BEFORE CONTINUING';
            INSTS.HEAD='                 INSTRUCTIONS';
            CALL WJS999('GWRITE',DCB,COUNT,INSTS,BUFADD);
            CALL CHECK;
            LITS='00000000000000000001'B;
            CALL WJS999('GCNTRL',DCB,'IND',LITS);
            CALL CHECK;
            INBIT='1'B;
            GO TO PFKEND;

/* TERMINATE INSTRUCTION DISPLAY */

TAG(18):CALL INIT;
            BUFFER.MARK=' ';
            IDENT=NULL;
            CALL SHOW;
            INBIT='0'B;
            GO TO PFKEND;

/* TERMINATE TREES */

TAG(31):WAIT_BIT='1'B;
```

```
        GO TO PFKEND;
SYSERR:BUFFER.PLINE='SYSTEM ERROR - TRY AGAIN';
        CALL SHOW;
PFKEND:TAG(3):TAG(29):TAG(9):TAG(28):TAG(13):TAG(14):TAG(15):TAG(19):
  TAG(20):TAG(21):TAG(22):TAG(23):TAG(24):TAG(25):TAG(26):TAG(27):
  TAG(12):TAG(30):;
        END;

 /* ATTENTION HANDLER FOR LIGHT PEN  */

        ON CONDITION(LP) BEGIN;
        DCL (K4 INIT(4096),K76 INIT(76),K150 INIT(150)) FIXED BIN(31);
        DCL LPEN(0:1) LABEL, (FTHR,P) POINTER, (FIF6 INIT(56),I,N,
          J) FIXED BIN;
        CALL WJS999('GCNTRL',DCB,'STR',BUFADD);
        CALL CHECK;
        BUFFER.PLINE=' ';
        GO TO LPEN(LPMODE);

/* IDENTIFY NODE TOUCHED BY LIGHT PEN */

LPEN(0):XLP=UNSPEC(ATTNAREA.XPOS);
        YLP=UNSPEC(ATTNAREA.YPOS);
        LNO=MAXLEV-(YLP+K150)/(K4/(MAXLEV+1))+1;
        RNO=(XLP+K76)/(K4/(WIDTH(LNO)+1));
        LEV=0;
        TIMES=0;
        IDENT=LOCATE(ROOT,LNO,RNO);
        CALL RESET(ROOT);
        IF IDENT=NULL THEN DO;
           BUFFER.MARK=' ';
           CALL SHOW;
           IDBIT='0'B;
           CALL INIT;
           GO TO LPEND;
        END;
        IDBIT='1'B;
        CALL INIT;
        XLP=(4096*RNO)/(WIDTH(LNO)+1)-FIF6;
        YLP=4096-((4096*LNO)/(MAXLEV+1));
        BUFFER.MARK='> <';
        BUFFER.XLOC1='01'B|SUBSTR(UNSPEC(XLP),17);
        BUFFER.YLOC1=SUBSTR(UNSPEC(YLP),17);
        CALL SHOW;
        GO TO LPEND;
```

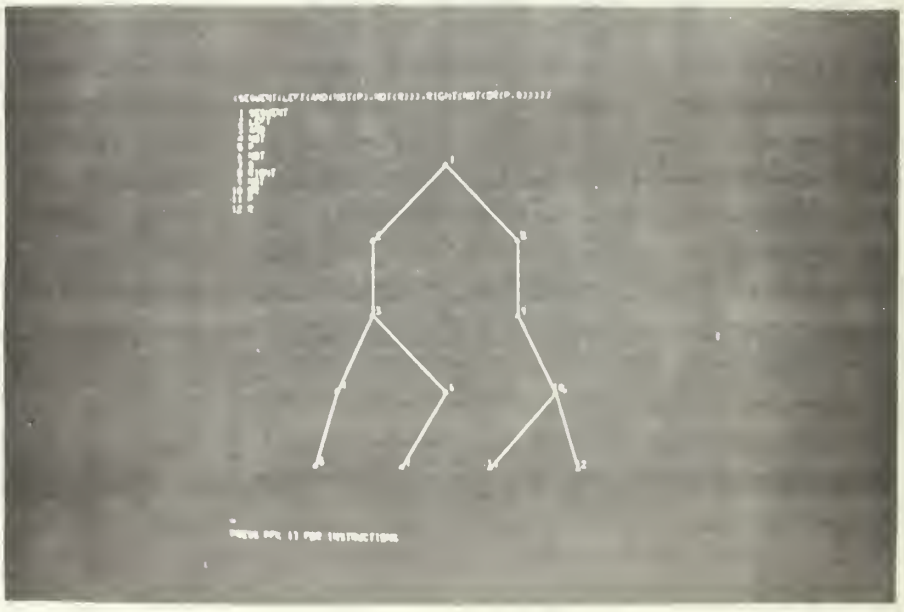
```
/* ADD STRUCTURE TO IDENT */

LPEN(1):CALL READ_CRT;
        BUFFER.MARK=' ';
        BUFFER.LINE=' ';
        BUFFER.PLINE=' ';
        XSTRT=XSTRT+DIST/2;
        XLP=UNSPEC(ATTNAREA.XPOS);
        DO I=1 TO IDENT->NODE.DEGREE;
           IF XLP<XSTRT+(I-1)*DIST THEN GO TO INS;
        END;
        I=IDENT->NODE.DEGREE+1;
        IF IDENT=TDT THEN GO TO TOP;
  INS:  FTHR=FATHER(TDT,IDENT);
        CALL RESET(TDT);
        IF FTHR=NULL THEN DO;
           BUFFER.PLINE='SYSTEM ERROR - TRY AGAIN';
           CALL DSPLY;
              GO TO LPEND;
        END;
 TOP:   P=ROOT;
        CALL ENDBLNK;
        IF SUBSTR(CARD,COL,1)='(' THEN DO;
           CALL PASS1;
           IF ROOT=NULL THEN DO;
              ROOT=P;
              GO TO LPEND;
           END;
        END; ELSE DO;
           N=SAVE_NO;
           IF N=0¬N>SAVE.NUM THEN DO;
              BUFFER.PLINE='TREE NOT FOUND IN SAVEAREA';
              BUFFER.LINE=' ';
              CALL SHOW;
              GO TO LPEND;
           END;
           ROOT=SAVE.ROOTS(N).PTR;
        END;
        IF IDENT=TDT THEN DO;
           TDT=ADD(IDENT,ROOT,I);
           ROOT=TDT;
           IF RTNO¬=0 THEN SAVE.ROOTS(RTNO).PTR=TDT;
        END; ELSE DO;
           J=NUMBER(FTHR,IDENT);
           FTHR->NODE.BRANCH(J)=ADD(IDENT,ROOT,I);
           IF IDENT=P THEN ROOT=FTHR->NODE.BRANCH(J); ELSE ROOT=P;
```

```
        END;
  SHW:  CALL DSPLY;
        IDBIT='0'B;
        BUFFER.MARK=' ';
        IDENT=NULL;
        CALL INIT;
LPEND:  LPMODE=0;
        END;

 /* ATTENTION HANDLER FOR END KEY */

        ON CONDITION(ENDCAN) BEGIN;  END;

/* INITIALIZE GRAPHICS PACKAGE */

        SPT=PTR;
        ALLOCATE SAVE;
        CALL WJS999('OPEN',DCB,DD);
        CALL CHECK;
        CALL WJS999('OCBP',WORKAREAZ,OACBZ,OCBPZ);
        CALL CHECK;
        CALL WJS999('XYLIM',XYLIMZ,LIMPARM);
        CALL CHECK;
        CALL WJS999('SPAR',DCB,ATTNAREA);
        CALL CHECK;
        IF CHSTR¬='' THEN BUFFER.PLINE=CHSTR;
        ELSE BUFFER.PLINE='PRESS PFK 17 FOR INSTRUCTIONS';

 /*  WAIT LOOP FOR ATTENTION FROM CRT  */
        ROOT,TDT=SPT;
        IF SPT=NULL THEN GO TO BRITE;
        BUFFER.GTRU=NOP4;
        CALL DSPLY;
        RTNO=0;
        IDENT=NULL;
        SVBIT='1'B;

BRITE:  CALL INIT;
        CALL SHOW;
LOPP:   DELAY(001000);
        IF ¬ WAIT_BIT THEN GO TO LOPP;

 /* TERMINATE PROGRAM  */

        CALL READ_CRT;
        IF CARD¬=' ' THEN DO;
```

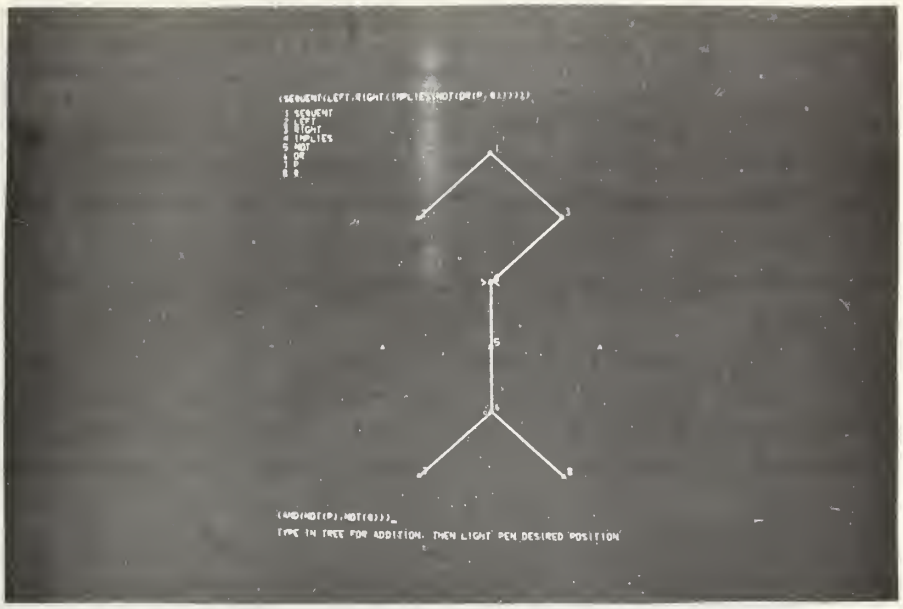
```
            COL=148;
            DO WHILE (SUBSTR(CARD,COL,1)=' ');
               COL=COL-1;
            END;
         I=1;
            DO WHILE (SUBSTR(CARD,I,1)=' ');
               I=I+1;
            IF I>=COL THEN GO TO FRE;
            END;
            CHSTR=SUBSTR(CARD,I,COL-I+1);
         END;
FRE:     FREE SAVE;
         CALL WJS999('GCNTRL',DCB,'HLT',BUFADD);
         CALL CHECK;
         LITS='0'B;
         CALL WJS999('GCNTRL',DCB,'IND',LITS);
         CALL CHECK;
         CALL WJS999('DAR',ATTNAREA);
         CALL CHECK;
         CALL WJS999('CLOSE',DCB);
         IF SPT=NULL THEN SPT=ROOT;
         RETURN (SPT);
         END TREEPAK;
```



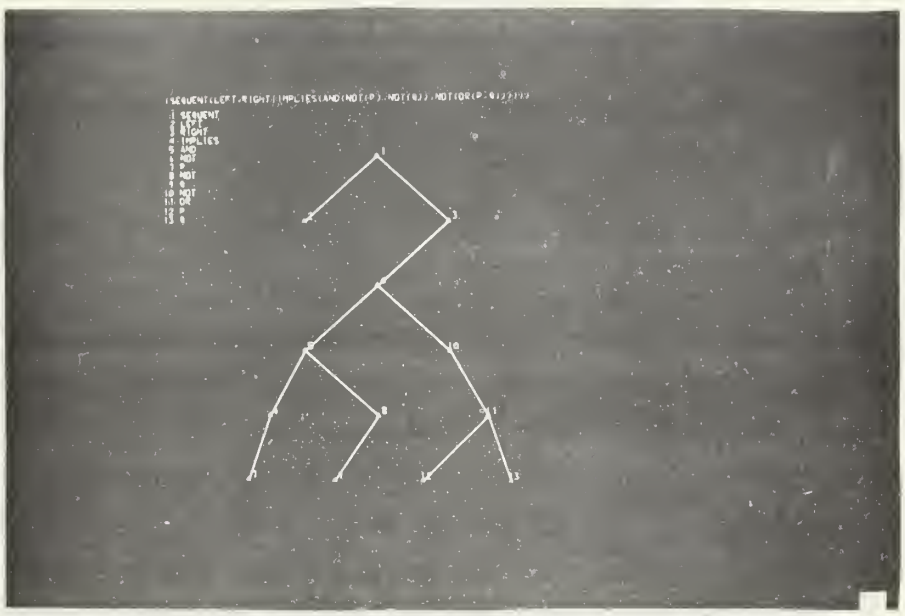
Tree representation of a sequent.



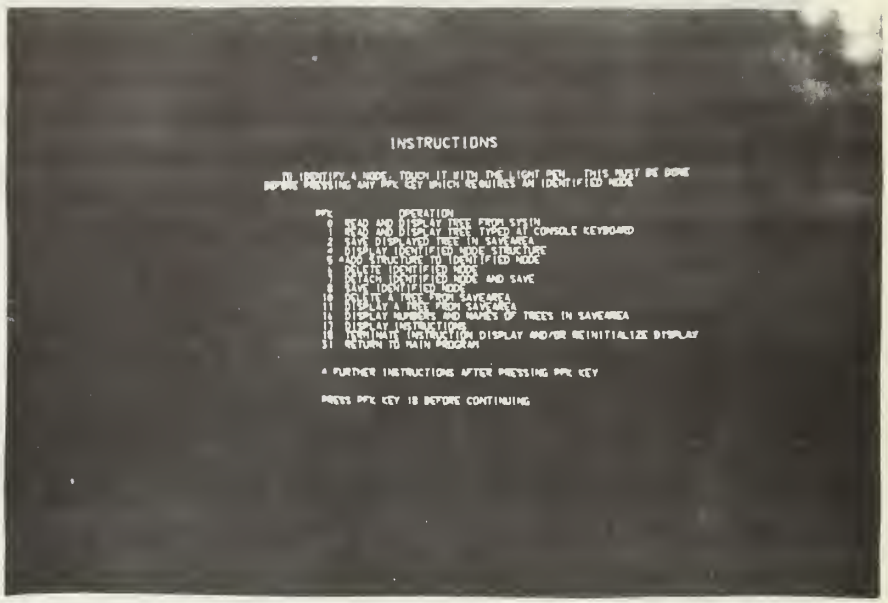
Result of removing "AND" from the preceding sequent.



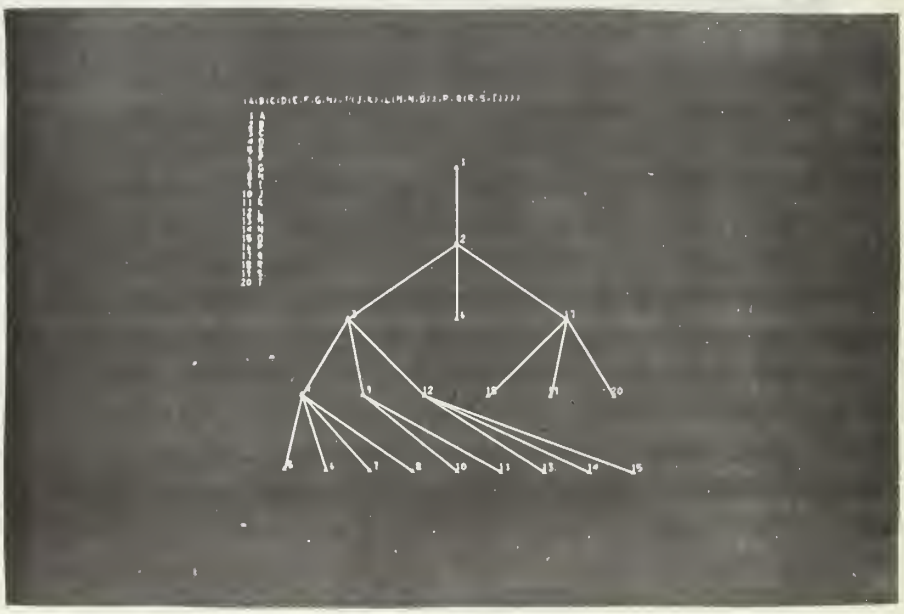
Node number 4 is the identified node. The PFK key for "add" has been pressed. The subtree to be added is typed at lower left.



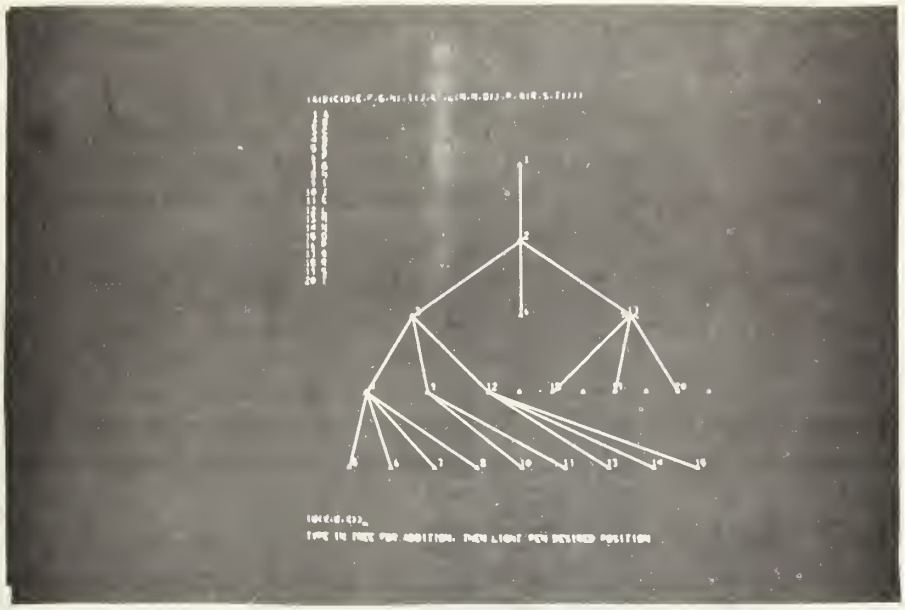
Result of touching the light pen to the left asterisk in the preceding tree.



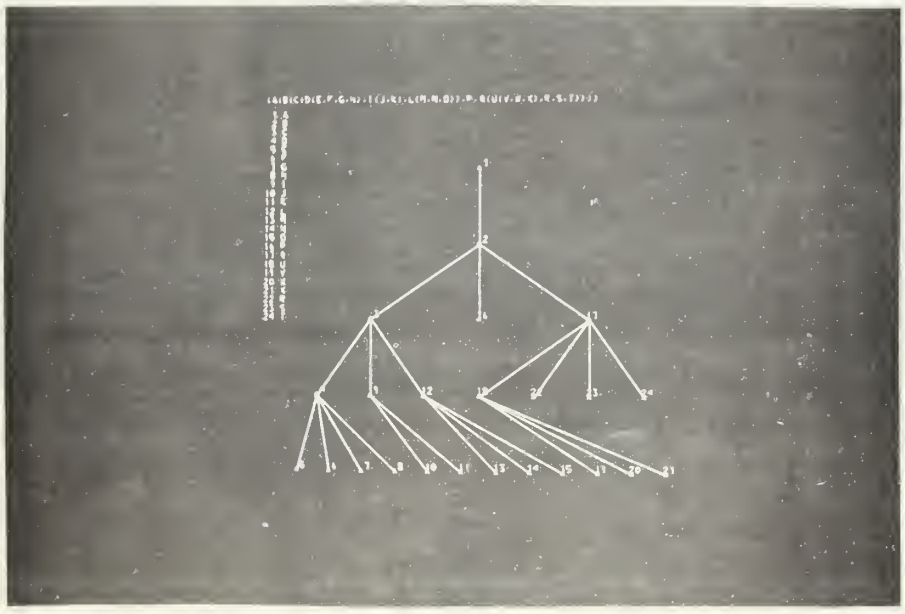
General instruction display.



A tree which was created at the console keyboard.



Node number 17 is the identified node. The "add" PFK key has been pressed.



After adding the new subtree to node 17 in the preceding tree.

```
//BOX#106C JOB (0373,0331PT,CSG8),' HOLIFIELD ',TYPRUN=HOLD
// EXEC PL1LFCLG,REGION.PL1L=100K,REGION.GO=150K
//PL1L.SYSIN DD *

 WANG:    PROC OPTIONS(MAIN);

 /* DECLARE VARIABLES */

          DCL I BIN FIXED, MSGVAL RETURNS (FIXED DECIMAL);
          DCL 1 NODE BASED(PTR),
               2 NAME CHAR(10),
               2 DEGREE FIXED BIN,
                 2 BRANCH(0:BOUND REFER(NODE.DEGREE)) POINTER;
          DCL BOUND FIXED BIN, (ADD,DELETE,TREEPAK) RETURNS(POINTER);
          DCL MSG CHAR(74) VARYING, RULE CHAR(4),STACK POINTER;
          DCL (CURSEQ,CURR,CURL,R) POINTER,FD FIXED DECIMAL;
          DCL OUP CHAR(148) VARYING, ONE FIXED BIN INIT(1);
          DCL COPY RETURNS (POINTER),PROCVAL RETURNS (FIXED BIN),
             NEXT_OP RETURNS (FIXED DECIMAL);
 /* THIS ROUTINE ADDS STRUCTURE POINTED TO BY R TO NODE POINTED TO BY Q
 IN POSITION N.  RETURNS POINTER TO NODE WHICH REPLACED Q */

 ADD:     PROCEDURE(Q,R,N) POINTER;
          DCL (Q,R,S) POINTER, (I,N) FIXED BIN;
          BOUND=Q->NODE.DEGREE+1;
          ALLOCATE NODE;
          NODE.NAME=Q->NODE.NAME;
          DO I=1 TO N-1;
             NODE.BRANCH(I)=Q->NODE.BRANCH(I);
          END;
          NODE.BRANCH(N)=R;
          DO I=N+1 TO NODE.DEGREE;
             NODE.BRANCH(I)=Q->NODE.BRANCH(I-1);
          END;
          FREE Q->NODE;
          RETURN (PTR);
          END ADD;

 /* THIS ROUTINE DELETES THE NTH BRANCH FROM THE NODE POINTED TO BY Q.
   RETURNS A POINTER TO THE NODE WHICH REPLACED Q. */

 DELETE: PROCEDURE(Q,N) POINTER;
          DCL Q POINTER, (N,I) FIXED BIN;
```

```
            BOUND=Q->NODE.DEGREE-1;
            ALLOCATE NODE;
            NODE.NAME=Q->NODE.NAME;
            DO I=1 TO N-1;
               NODE.BRANCH(I)=Q->NODE.BRANCH(I);
            END;
            DO I=N+1 TO Q->NODE.DEGREE;
               NODE.BRANCH(I-1)=Q->NODE.BRANCH(I);
            END;
            FREE Q-> NODE;
            RETURN (PTR);
            END DELETE;

/* ROUTINE TO EVALUATE MESSAGE FROM USER */

 MSGVAL: PROC FIXED DECIMAL;
            DCL I FIXED DECIMAL, COL FIXED BIN;
            IF MSG='' THEN RETURN (1);
            COL=1;
            DO WHILE (COL<=LENGTH(MSG));
               IF UNSPEC(SUBSTR(MSG,COL,1))<UNSPEC('0') THEN RETURN(1);
               COL=COL+1;
            END;
            I=MSG;
            RETURN(I);
            END MSGVAL;

/* ROUTINE TO BUILD THE OUTPUT STRING */

BLD_STR:PROC(Q) RECURSIVE;
            DCL Q POINTER,I FIXED BIN;
            DCL COL FIXED BIN;
            COL=10;
            DO WHILE (SUBSTR(Q->NODE.NAME,COL,1)=' ');
               COL=COL-1;
            END;
            OUP=OUP||SUBSTR(Q->NODE.NAME,1,COL);
            DO I=1 TO Q->NODE.DEGREE;
               IF I=1 THEN OUP=OUP||'('; ELSE OUP=OUP||',';
               CALL BLD_STR((Q->NODE.BRANCH(I)));
               IF I=Q->NODE.DEGREE THEN OUP=OUP||')';
            END;
            RETURN;
            END BLD_STR;

/* ROUTINE TO TEST THE CONNECTIVE FREE SEQUENT */
```

```
PROCVAL:PROC(Q) FIXED BIN;
        DCL (S,T,R,L,Q) POINTER, (I,J) FIXED BIN;
        DCL (ONE INIT(1),ZERO INIT(0)) FIXED BIN;
        R=Q->NODE.BRANCH(2);
        L=Q->NODE.BRANCH(1);
        DO I=1 TO L->NODE.DEGREE;
           S=L->NODE.BRANCH(I);
           DO J=1 TO R->NODE.DEGREE;
              T=R->NODE.BRANCH(J);
              IF S->NODE.NAME=T->NODE.NAME THEN DO;
                 PUT EDIT('VALID') (SKIP,A,SKIP(2));
                 RETURN (ONE);
              END;
           END;
        END;
        PUT EDIT ('NOT VALID') (SKIP,A,SKIP(2));
        RETURN(ZERO);
        END PROCVAL;

/* ROUTINE TO FIND AND ELIMINATE THE NEXT CONNECTIVE */

NEXT_OP:PROC(S) FIXED DECIMAL;
        DCL (L,T,S) POINTER, I FIXED BIN;
        T=S->NODE.BRANCH(1);
OPER:   DO I=1 TO T->NODE.DEGREE;
           L=T->NODE.BRANCH(I);
           IF L->NODE.NAME='EQUIV' THEN DO;
              CALL PROCB(T->NODE.NAME,I);
           GO TO RET;
        END;
           IF L->NODE.NAME='AND' THEN DO;
              CALL PROCC(T->NODE.NAME,I);
              GO TO RET;
           END;
           IF L->NODE.NAME='OR' THEN DO;
              CALL PROCD(T->NODE.NAME,I);
              GO TO RET;
           END;
           IF L->NODE.NAME='NOT' THEN DO;
              CALL PROCF(T->NODE.NAME,I);
              GO TO RET;
           END;
           IF L->NODE.NAME='IMPLIES' THEN DO;
              CALL PROCI(T->NODE.NAME,I);
              GO TO RET;
```

```
            END;
         END;
         IF T->NODE.NAME='LEFT' THEN DO;
         T=S->NODE.BRANCH(2);
            GO TO OPER;
         END;
         RETURN(0);
RET:     RETURN(1);
         END NEXT_OP;

/* ROUTINE TO FREE STORAGE NO LONGER NEEDED */

RID:     PROC(P) RECURSIVE;
         DCL P POINTER, I FIXED BIN;
         DO I=1 TO P->NODE.DEGREE;
            CALL RID((P->NODE.BRANCH(I)));
         END;
         FREE P->NODE;
         RETURN;
         END RID;

/* ROUTINE TO MAKE A COPY OF A TREE */

COPY:    PROC(P) POINTER RECURSIVE;
         DCL (Q,P) POINTER, I FIXED BIN;
         BOUND=P->NODE.DEGREE;
         ALLOCATE NODE;
         Q=PTR;
         NODE.NAME=P->NODE.NAME;
         DO I=1 TO P->NODE.DEGREE;
            Q->NODE.BRANCH(I)=COPY((P->NODE.BRANCH(I)));
         END;
         RETURN(Q);
         END COPY;

/* ROUTINE TO ELIMINATE EQUIV */

PROCB:   PROC(CH,I);
         DCL CH CHAR(10), I FIXED BIN,(NEW,R,S,Q) POINTER;
         IF CH='RIGHT' THEN DO;
            NEW=COPY(CURSEQ);
            Q=CURR->NODE.BRANCH(I);
            CURR->NODE.BRANCH(I)=Q->NODE.BRANCH(1);
            CURSEQ->NODE.BRANCH(1)=ADD(CURL,(Q->NODE.BRANCH(2)),ONE);
            CURL=CURSEQ->NODE.BRANCH(1);
            R=NEW->NODE.BRANCH(2);
```

```
            S=R->NODE.BRANCH(I);
            R->NODE.BRANCH(I)=S->NODE.BRANCH(2);
            R=NEW->NODE.BRANCH(1);
            NEW->NODE.BRANCH(1)=ADD(R,S->NODE.BRANCH(1),ONE);
            STACK=ADD(STACK,NEW,ONE);
            FREE S->NODE,Q->NODE;
            RULE='6A';
            RETURN;
         END;
         IF CH='LEFT' THEN DO;
            NEW=COPY(CURSEQ);
            Q=CURL->NODE.BRANCH(I);
            CURSEQ->NODE.BRANCH(1)=DELETE(CURL,I);
            CURL=CURSEQ->NODE.BRANCH(1);
            CURSEQ->NODE.BRANCH(2)=ADD(CURR,Q->NODE.BRANCH(1),CURR->NODE
.DEGREE+ONE);
            CURR=CURSEQ->NODE.BRANCH(2);
            CURSEQ->NODE.BRANCH(2)=ADD(CURR,Q->NODE.BRANCH(2),CURR->NODE
.DEGREE+ ONE);
            CURR=CURSEQ->NODE.BRANCH(2);
            R=NEW->NODE.BRANCH(1);
            S=R->NODE.BRANCH(I);
            NEW->NODE.BRANCH(1)=DELETE(R,I);
            R=NEW->NODE.BRANCH(1);
            NEW->NODE.BRANCH(1)=ADD(R,S->NODE.BRANCH(2),ONE);
            R=NEW->NODE.BRANCH(1);
            NEW->NODE.BRANCH(1)=ADD(R,S->NODE.BRANCH(1),ONE);
            STACK=ADD(STACK,NEW,ONE);
            FREE S->NODE,Q->NODE;
            RULE='6B';
            RETURN;
         END;
         END PROCB;

/* ROUTINE TO ELIMINATE IMPLIES */

PROCI:   PROC(CH,I);
         DCL CH CHAR(10), I FIXED BIN, (NEW,Q,R,S) POINTER;
         IF CH='RIGHT' THEN DO;
            Q=CURR->NODE.BRANCH(I);
            CURR->NODE.BRANCH(I)=Q->NODE.BRANCH(2);
            CURSEQ->NODE.BRANCH(1)=ADD(CURL,Q->NODE.BRANCH(1),CURL->NODE
.DEGREE+ ONE);
            CURL=CURSEQ->NODE.BRANCH(1);
            FREE Q->NODE;
            RULE='5A';
```

```
            RETURN;
        END;
        IF CH='LEFT' THEN DO;
            Q=CURL->NODE.BRANCH(I);
            NEW=COPY(CURSEQ);
            CURL->NODE.BRANCH(I)=Q->NODE.BRANCH(2);
            R=NEW->NODE.BRANCH(1);
            S=R->NODE.BRANCH(I);
            CALL RID(S->NODE.BRANCH(2));
            NEW->NODE.BRANCH(1)=DELETE(R,I);
            R=NEW->NODE.BRANCH(2);
            NEW->NODE.BRANCH(2)=ADD(R,S->NODE.BRANCH(1),R->NODE.DEGREE+
ONE);
            STACK=ADD(STACK,NEW,ONE);
            FREE Q->NODE,S->NODE;
            RULE='5B';
            RETURN;
        END;
        END PROCI;

/* ROUTINE TO ELIMINATE OR */

PROCD:  PROC(CH,I);
        DCL CH CHAR(10),I FIXED BIN,(Q,NEW,R,S) POINTER;
        IF CH='LEFT' THEN DO;
            Q=CURL->NODE.BRANCH(I);
            NEW=COPY(CURSEQ);
            CURL->NODE.BRANCH(I)=Q->NODE.BRANCH(1);
            R=NEW->NODE.BRANCH(1);
            S=R->NODE.BRANCH(I);
            R->NODE.BRANCH(I)=S->NODE.BRANCH(2);
            CALL RID(S->NODE.BRANCH(1));
            FREE Q->NODE,S->NODE;
            RULE='4B';
            RETURN;
        END;
        IF CH='RIGHT' THEN DO;
            Q=CURR->NODE.BRANCH(I);
            CURR->NODE.BRANCH(I)=Q->NODE.BRANCH(1);
            CURSEQ->NODE.BRANCH(2)=ADD(CURR,Q->NODE.BRANCH(2),I+ONE);
            CURR=CURSEQ->NODE.BRANCH(2);
            FREE Q->NODE;
            RULE='4A';
            RETURN;
        END;
        END PROCD;
```

```
/* ROUTINE TO ELIMINATE NOT */

PROCF:   PROC(CH,I);
         DCL CH CHAR(10), I FIXED BIN, Q POINTER;
         IF CH='LEFT' THEN DO;
            Q=CURL->NODE.BRANCH(I);
            CURSEQ->NODE.BRANCH(2)=ADD(CURR,Q->NODE.BRANCH(1),CURR->NODE
.DEGREE + ONE);
            CURR=CURSEQ->NODE.BRANCH(2);
            CURSEQ->NODE.BRANCH(1)=DELETE(CURL,I);
            CURL=CURSEQ->NODE.BRANCH(1);
            FREE Q->NODE;
            RULE='2B';
            RETURN;
         END;
         IF CH='RIGHT' THEN DO;
            Q=CURR->NODE.BRANCH(I);
            CURSEQ->NODE.BRANCH(1)=ADD(CURL,Q->NODE.BRANCH(1),ONE);
            CURL=CURSEQ->NODE.BRANCH(1);
            CURSEQ->NODE.BRANCH(2)=DELETE(CURR,I);
         CURR=CURSEQ->NODE.BRANCH(2);
            FREE Q->NODE;
            RULE='2A';
            RETURN;
         END;
         END PROCF;

/* ROUTINE TO ELIMINATE AND */

PROCC:   PROC(CH,I);
         DCL CH CHAR(10), I FIXED BIN,(R,S,Q,NEW) POINTER;
         IF CH='LEFT' THEN DO;
            Q=CURL->NODE.BRANCH(I);
            CURL->NODE.BRANCH(I)=Q->NODE.BRANCH(1);
            CURSEQ->NODE.BRANCH(1)=ADD(CURL,Q->NODE.BRANCH(2),I+ONE);
            CURL=CURSEQ->NODE.BRANCH(1);
            FREE Q->NODE;
            RULE='3B';
            RETURN;
         END;
         IF CH='RIGHT' THEN DO;
            NEW=COPY(CURSEQ);
            Q=CURR->NODE.BRANCH(I);
            CURR->NODE.BRANCH(I)=Q->NODE.BRANCH(1);
            R=NEW->NODE.BRANCH(2);
```

```
                S=R->NODE.BRANCH(I);
                R->NODE.BRANCH(I)=S->NODE.BRANCH(2);
                STACK=ADD(STACK,NEW,ONE);
                CALL RID(S->NODE.BRANCH(1));
                FREE S->NODE,Q->NODE;
                RULE='3A';
                RETURN;
            END;
            END PROCC;

/* CENTRAL LOOP TO CALL ROUTINES AND CALL TREEPAK */

            BOUND=0;
            ALLOCATE NODE SET (STACK);
            STACK->NODE.NAME='STACK';
            RULE=' ';
            PUT LIST ('              THEOREM PROOF BY THE WANG METHOD') SKIP;
            PUT LIST ('RULE       LINES OF PROOF') SKIP(4);
            MSG='READY FOR THEOREM INPUT';
            R=NULL;
LOOP:       CURSEQ=TREEPAK(R,MSG);
            IF MSG='END' THEN GO TO STOP;
            IF CURSEQ->NODE.DEGREE<2 THEN GO TO STOP;
            CURR=CURSEQ->NODE.BRANCH(2);
            CURL=CURSEQ->NODE.BRANCH(1);
            DO I=1 TO MSGVAL;
                OUP='(';
                CALL BLD_STR(CURSEQ);
                OUP=OUP||T')';
                PUT LIST (RULE||OUP) SKIP;
                IF NEXT_OP(CURSEQ)=0 THEN GO TO EVAL;
            END;
            R=CURSEQ;
            MSG='';
            GO TO LOOP;
 EVAL:      IF PROCVAL(CURSEQ)=0 THEN DO;
                PUT LIST ('NEW THEOREM') SKIP(4);
                MSG='THEOREM IS NOT VALID, READY FOR INPUT';
            R=NULL;
            RULE=' ';
                GO TO LOOP;
            END;
            IF STACK->NODE.DEGREE>0 THEN DO;
                R=CURSEQ;
                CURSEQ=STACK->NODE.BRANCH(STACK->NODE.DEGREE);
                CURR=CURSEQ->NODE.BRANCH(2);
```

```
            CURL=CURSEQ->NODE.BRANCH(1);
            STACK=DELETE(STACK,(STACK->NODE.DEGREE));
            CALL RID(R);
            RULE=' ';
         R=CURSEQ;
         MSG='';
            GO TO LOOP;
         END;
         MSG='THEOREM IS VALID, READY FOR INPUT';
         R=NULL;
         PUT LIST ('NEW THEOREM') SKIP(4);
         RULE=' ';
         GO TO LOOP;
STOP:STOP;
         END WANG;
/*

//LKED.LIBA DD DSN=S0373.PLIGPAK,VOL=SER=LINDA,UNIT=2314,DISP=SHR
//LKED.SYSIN DD *
         LIBRARY LIBA(WJS999,TREEPAK)
 /*
//GO.SYSPRINT  DD  DCB=BLKSIZE=133
//GO.DISPLAY  DD  UNIT=(2250-1)
//GO.PLIDUMP DD SYSOUT=A,SPACE=(TRK,4)
//GO.SYSIN DD *
(SEQUENT(LEFT,RIGHT(IMPLIES(AND(NOT(P),NOT(Q)),EQUIV(P,Q)))))))))))))))))))))

(SEQUENT(LEFT,RIGHT(IMPLIES(NOT(OR(P,Q)),NOT(P)))))))))))))))))))))

(SEQUENT(LEFT,RIGHT(IMPLIES(IMPLIES(P,NOT(P)),NOT(P)))))))))))))))))))))

(SEQUENT(LEFT,RIGHT(IMPLIES(OR(P,OR(Q,R)),OR(OR(P,Q),R)))))))))))))))))
```

LIST OF REFERENCES

1. Knuth, D. E., Fundamental Algorithms, v. 1, Addison Wesley, 1968.

2. Lawson, H. W., "PL/I List Processing," Communications of the ACM, v. 10, p. 358-367, June 1967.

3. Londe, D. L. and Schoene, W. J., "TGT: Transformational Grammer Tester," AFIPS Conference Proceedings, SJCC, Spartan, 1968.

4. Air Force Cambridge Laboratories Report 67-0167, CRT-Aided Semi-Automated Mathematics, by J. H. Bennet and others, Applied Logic Corporation, January 1967.

5. International Business Machines Contributed Program Library No. 360D-00.6.009, PL/I Graphics Subroutine Package, by William Speary, August 1968.

6. International Business Machines Form C27-6909-4, Graphic Programming Services For IBM 2250 Display Unit, 1967.

7. International Business Machines Form C28-8201, PL/I Reference Manual, 1968.

8. International Business Machines Form C28-6594-3, PL/I (F) Programmer's Guide, 1967.

9. Wang, H., "Toward Mechanical Mathematics," IBM Journal, v. 4, January 1960.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	20
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Commandant of the Marine Corps (Code AO3C) Headquarters, U. S. Marine Corps Washington, D. C. 20380	1
4.	James Carson Breckinridge Library Marine Corps Development and Educational Command Quantico, Virginia 22134	1
5.	LT(jg) G. A. Kildall, Instructor, Code 53 Kd Department of Mathematics Naval Postgraduate School Monterey, California 93940	1
6.	Asst. Professor G. L. Barksdale, Code 53 Bv Department of Mathematics Naval Postgraduate School Monterey, California 93940	1
7.	Professor D. G. Williams, Code 0211 Computing Center Naval Postgraduate School Monterey, California 93940	1
8.	Major C. M. Holifield 702 Smith Street Marion, Alabama 36756	1
9.	Asst. Professor G. E. Heidorn, Code 55 Hd Department of Operations Analysis Naval Postgraduate School Monterey, California 93940	1

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

Naval Postgraduate School
Monterey, California 93940

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

3. REPORT TITLE

Graphic Display and Manipulation of Tree-Type Data Structures

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Master's Thesis; December 1969

5. AUTHOR(S) *(First name, middle initial, last name)*

Claude M. Hollifield, Jr.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
December 1969	81	9

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Naval Postgraduate School Monterey, California 93940

13. ABSTRACT

The subroutine package TREEPAK is designed to interface a PL/I program with a graphic display unit, allowing display, creation and manipulation of tree structures, while providing the user with a graphic representation of the tree. The package is implemented under IBM System/360 using the IBM 2250-1 graphic display unit as the primary input/output device. Facilities include creation of trees from the alphanumeric keyboard and cards, adding and deleting subtrees in existing trees, and saving and retrieving trees. Trees may be passed between TREEPAK and the user written main program. An implementation of the Wang Algorithm is given as an example application program.

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
graphic display of trees						
tree-type data structures						